NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

DFQL:
A GRAPHICAL DATAFLOW
QUERY LANGUAGE

by

Gard J. Clark

September 1991

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited.

T257790

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified		1b. RESTRICTIVE MARKINGS	
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/ AVAILABILITY OF REPORT Approved for public release; distribution is unlimited.	
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)	
6a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School	6b. OFFICE SYMBOL *(If Applicable)* 37	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School	
6c. ADDRESS *(city, state, and ZIP code)* Monterey, CA 93943-5000		7b. ADDRESS *(city, state, and ZIP code)* Monterey, CA 93943-5000	
8a. NAME OF FUNDING/SPONSORING ORGANIZATION	6b. OFFICE SYMBOL *(If Applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER	
8c. ADDRESS *(city, state, and ZIP code)*		10. SOURCE OF FUNDING NUMBERS	
		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
DFQL: A GRAPHICAL DATAFLOW QUERY LANGUAGE

12. PERSONAL AUTHOR(S)
Gard J. Clark

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(year, month,day)*	15. PAGE COUNT
Master's Thesis	FROM TO	1991 September	310

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17. COSATI CODES			18. SUBJECT TERMS *(continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUBGROUP	database, query language, dataflow programming, object-oriented programming, relational model, SQL, human factors

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

In nearly all large organizations, the Navy and Department of Defense being no exceptions, the use of database management systems (DBMS's) has become widespread. The prevailing data model for modern DBMS's is the relational model developed by Codd in the early 1970's. The relational model's superiority is due to its well thought out design and founding in mathematical logic. The de facto standard query language for relational DBMS's is IBM's Structured Query Language (SQL). Although SQL is the most widely used query language today, it has many problems, especially in the ease-of-use area.

The purpose of this thesis is to design, implement, and test a new query language, *DFQL*, which will mitigate SQL's ease-of-use problems. DFQL provides a graphical query interface based on the dataflow paradigm in order to allow a user to easily and incrementally construct queries for a relational database. DFQL is relationally complete, maintains relational operational closure, and is designed to be easily extensible by the end user. DFQL has been implemented on an Apple Macintosh using an ORACLE relational DBMS. A simple human factors experiment was performed in which DFQL's ease of query writing compared favorably to that of SQL.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
[X] UNCLASSIFIED/UNLIMITED [ ] SAME AS RPT. [ ] DTIC USERS	Unclassified	
22a. NAME OF RESPONSIBLE INDIVIDUAL C. Thomas Wu	22b. TELEPHONE (Include Area Code) (408) 646-3391	22c. OFFICE SYMBOL CS/Wq

Approved for public release; distribution is unlimited.

DFQL:
A Graphical Dataflow
Query Language

by

Gard J. Clark
Lieutenant, United States Navy
B.S., United States Naval Academy, 1985

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September, 1991

ABSTRACT

In nearly all large organizations, the Navy and Department of Defense being no exceptions, the use of database management systems (DBMS's) has become widespread. The prevailing data model for modern DBMS's is the relational model developed by Codd in the early 1970's. The relational model's superiority is due to its well thought out design and founding in mathematical logic. The de facto standard query language for relational DBMS's is IBM's Structured Query Language (SQL). Although SQL is the most widely used query language today, it has many problems, especially in the ease-of-use area.

The purpose of this thesis is to design, implement, and test a new query language, *DFQL*, which will mitigate SQL's ease-of-use problems. DFQL provides a graphical query interface based on the dataflow paradigm in order to allow a user to easily and incrementally construct queries for a relational database. DFQL is relationally complete, maintains relational operational closure, and is designed to be easily extensible by the end user. DFQL has been implemented on an Apple Macintosh using an ORACLE relational DBMS. A simple human factors experiment was performed in which DFQL's ease of query writing compared favorably to that of SQL.

1 2013
C4/8135
c.1

TABLE OF CONTENTS

I. INTRODUCTION 1

A. MOTIVATION 1

B. OBJECTIVES OF A VISUAL DATABASE INTERFACE 2

C. THESIS OVERVIEW 3

II. PREVIOUS WORK 5

A. GENERAL DISCUSSION 5

B. PROBLEMS WITH SQL 6

1. Basis of SQL 6

2. Declarative Nature 7

3. Universal Quantification 8

4. Lack of Orthogonality 10

5. Nesting Construct 11

6. Lack of Functional Notation 12

7. Other Issues 13

C. EXISTING VISUAL QUERY LANGUAGES 13

1. Forms Based Interfaces 14

a. STBE--Summary Table By Example 16

b. AQL--A Query Language 18

c. RC/S--Relational Calculus/Sets 18

d. Objectives, Benefits, and Drawbacks 19

2. Entity-Relationship Model Interface 20

a. GQL/Andyne--Graphical Query Language 23

b. GDML--Graphical Data Manipulation Language 24

c. QBD*--Query By Diagram* 26

d. GUIDE--Graphical User Interface for Database Exploration . 27

e. GRAQULA--Graphical Query Language 28

f. Objectives, Benefits, and Drawbacks 32

3. Other Approaches 33

a. PICASSO--Picture Aided Sophisticated Sketch Of Database Queries 33

b. IFO and SNAP--A Graphics-based Schema Manager 36

D. DATAFLOW PROGRAMMING LANGUAGES 38

1. Dataflow Diagram Description 38

2. Visual Dataflow Programming 39

III. DESCRIPTION OF DFQL 42

A. CONCEPT 42

1. DFQL Operators 43

a. Basic Operators 45

(1) Select. 47

(2) Project. 48

(3) Join. 49

(4) Union. 52

(5) Diff. 53

(6) Groupcnt. 54

b. Other Primitives 56

(1) Eqjoin. 56

(2) GroupALLsatisfy. 58

(3) GroupNsatisfy. 59

(4) Aggregate operators. 60

(5) Intersect. 61

c. Display Operators 61

(1) DISPLAY. 62

(2) SDISPLAY. 62

d. User-Defined Operators 64

2. DFQL Query Construction 68

a. Incremental Queries 69

b. Universal Quantification 73

c. Nesting and Functional Notation 73

d. Graph Structure of DFQL Query 74

B. USER INTERFACE FOR DFQL 75

1. Starting The Program 76
2. DB INTERFACE Window Items 76
a. Buttons 76
b. Drawing Area 80
3. Query Results Window 82
4. Menu Items 84
a. Apple 85
b. File 85
c. Edit 87
d. Primitives 89
e. UserOps 89
f. Options... 94
g. Info 95
h. Special... 95
C. IMPLEMENTATION OF DFQL 97
1. Prograph -- Object-Oriented Dataflow Language 97
a. Prograph Code 98
b. Object-Oriented Features 104
2. DFQL Implementation Strategy 109
a. User Interface to Stored Query Graph 111
a. Query Graph to SQL 113
b. SQL to Query Display 119

1. Goals of the DFQL Interpreter Class Structure 120

IV. ANALYSIS OF DFQL 122

A. HUMAN FACTORS ANALYSIS OF QUERY LANGUAGES 122

1. Testing for Ease-of-Use 122

2. Applicable Results of Previous Human Factors Studies 124

B. EXPERIMENTAL COMPARISON OF DFQL WITH SQL 126

1. Assessment of the Experiment 126

a. Subjects 126

b. Teaching Method 127

c. Kinds of Tasks 127

d. Test Questions 128

e. Test Environment 129

f. Evaluation Method 129

g. Experimenter Attitude 130

2. Experiment Results 130

3. Experiment Conclusions 133

C. ADVANTAGES OF DFQL 134

1. Power 134

2. Extensibility 134

3. Ease-Of-Use 135

a. Dataflow Representation 135

b. Orthogonality 135
c. Incremental Query Formulation and Execution 136
4. Visual Interface 136
D. SHORTCOMINGS OF THE DFQL CONCEPT 137
1. Interface Problems 137
2. Language Problems 138

V. CONCLUSIONS 140
A. REVIEW OF THE RESEARCH 140
B. FUTURE RESEARCH 140
1. Implementation Enhancement 141
2. Theoretical Investigation 142
C. SUMMARY 144

LIST OF REFERENCES 145
APPENDIX A. EXAMPLE DATABASE 148
APPENDIX B. HUMAN FACTORS EXPERIMENT DATA 151
APPENDIX C. DFQL SOURCE CODE 158
BIBLIOGRAPHY 293
INITIAL DISTRIBUTION LIST 294

LIST OF FIGURES

Figure 1. Example of QBE Query 14

Figure 2. Example of ER Diagram 21

Figure 3. Example Join in GRAQULA 30

Figure 4. Example PICASSO Hypergraph 35

Figure 5. Example Dataflow Diagram 39

Figure 6. Dataflow Program Fragment 40

Figure 7. Operator Construction 43

Figure 8. DFQL Basic Operators 46

Figure 9. Text Object 47

Figure 10. Example DFQL Select 48

Figure 11. Example DFQL Project 49

Figure 12. Example DFQL Join 51

Figure 13. Example DFQL Union 53

Figure 14. Example DFQL Diff 54

Figure 15. Example DFQL Groupcnt 55

Figure 16. Other DFQL Primitives 57

Figure 17. Example DFQL GroupALLsatisfy 59

Figure 18. Example DFQL GroupNsatisfy 59

Figure 19. Example DFQL Groupmax 61

Figure 20. Example DFQL SDISPLAY 63

Figure 21. DFQL Select - Project Query 64

Figure 22. Creating a User-Defined Operator 65

Figure 23. User-Defined Groupallsatisfy 67

Figure 24. Complete DFQL Query 69

Figure 25. Incremental Query Construction 71

Figure 26. Incremental Query Execution 72

Figure 27. Use of Multiple Display Operators 72

Figure 28. DFQL Main Window 77

Figure 29. Operator Creation 78

Figure 30. Text Object Creation 79

Figure 31. Example Select Operator Help 81

Figure 32. Query Results Window 83

Figure 33. DFQL Menu Bar 84

Figure 34. File Menu 85

Figure 35. Open... Dialog Box 86

Figure 36. Edit Menu 87

Figure 37. Primitives Menu 89

Figure 38. UserOps Menu 90

Figure 39. User Operator Definition Window 91

Figure 40. User-Defined Operator Selection 92

Figure 41. View User Operator Window 93

Figure 42. Options... Menu 94
Figure 43. Info Menu 95
Figure 44. Table Information 96
Figure 45. Starting the SQL*Plus Interpreter 96
Figure 46. Specifying Order of Execution 100
Figure 47. Prograph Case Structure 102
Figure 48. Iteration Over a List 103
Figure 49. Simple Iteration 104
Figure 50. Prograph System Classes 105
Figure 51. Attribute Window 106
Figure 52. Method Window 107
Figure 53. Method Referencing 108
Figure 54. Block Structure of DFQL Interpreter 110
Figure 55. Interface to Object Representation 112
Figure 56. Graph to SQL 113
Figure 57. Doallops 114
Figure 58. Adbopr/exeobj 116
Figure 59. Dbops/project 118
Figure 60. SQL to Result 119
Figure 61. Experiment Results 132

ACKNOWLEDGMENTS

I sincerely thank all of the people who assisted me in the conception and implementation of this thesis.

I am especially indebted to the ADP Division at Los Alamos National Laboratory for both their technical and logistical support. I particularly wish to thank Jim Hall, Pam French, Frank Welch, Joe Zowin, Bruce Panowski, Leann Anderson, Ken Sinclair, Lee Ankeny (C-9), and the ADP-1 Micro Support staff for all of their help and guidance. Special thanks go also to Doug Lier (A-6) whose support was invaluable.

I also would like to acknowledge Kevin Fontes for introducing me to both Prograph and the "intricacies" of Macintosh user-friendliness. Thanks also are due the technical support staff of TGSS for their assistance in implementation of this project. I greatly appreciate all of the guidance in the conception and implementation of DFQL by my advisor Dr. C. Thomas Wu.

Finally, I wish to thank my wife Beth for her patience and sacrifice in supporting me in this endeavor. Without her constant love and support, none of this would have been possible.

I. INTRODUCTION

A. MOTIVATION

The relational model for database management was introduced in 1969 by E. F. Codd (Codd, 1990, p. v). Compared to other actually implemented models, namely the hierarchical and network models, the relational model has the simplest and most uniform data structures and is founded much more rigorously in mathematics (Elmasri, 1990, p. 135). These features of the relational model make it a superior tool for most database implementations. In fact, the relational model has been labeled as "...one of the few pinnacles of computer science...." (Beech, 1989, p. 29)

The de facto standard query language associated with the relational model is Structured Query Language (SQL) invented by IBM (Chamberlin, 1974). There are serious problems with both the design and implementation of SQL as a query language that inhibit it in allowing the user easy access to the information stored in his relational database. Several of these problems are discussed by Codd in a two part article "Fatal Flaws in SQL" (Codd, 1988) and again in (Codd, 1990, chpt. 23). Our research primarily addresses what Codd terms as "The Psychological Mix-up" or human factors aspect of the language (Codd, 1990, pp. 379-382). We extend from his concern about the defined type of nesting in SQL to other psychological, or ease of use, issues. An example of another one of these issues is the difficulty in expressing the idea of universal quantification in

SQL (Negri, 1989). In general, we believe that a new database query language is required to allow users to achieve the maximum utility from the relational model.

B. OBJECTIVES OF A VISUAL DATABASE INTERFACE

Because of the problems associated with the current text based query languages (SQL in particular), we have proposed, designed, and implemented a graphical/visual interface to the relational model based on a dataflow paradigm. This DataFlow Query Language (DFQL) retains all of the power of current query languages and is equipped with an easy to use facility for extending the language with advanced operators, thus providing query facilities beyond the benchmark of first-order predicate logic. DFQL meets the following goals for a visual database interface which have been presented previously in other papers (IBM, 1991, pp. 1-2; Angelaccio, 1990, p. 1150):

- Employ a fully graphical environment as a user-friendly interface to the database
- Sufficient expressive power and functionality, including relational completeness
- Ease of use in learning, remembering, writing and reading the language's constructs
- Consistency, predictability, and naturalness (in both syntax and function)
- Simplicity and conciseness of features
- Clarity of definition, lack of ambiguity
- Ability to modify existing queries to form new queries incrementally
- High probability that users will write error-free queries
- Operator extensibility -- allow the user to create new operators in terms of existing ones, analogous to defining a function in a programming language

Examples of the approaches taken in DFQL to implement the above goals are:

- Complete faithfulness to relational algebra, especially in the requirement for operational closure
- Elimination of range variables from queries
- Elimination of nesting in query construction

There have been several research efforts directed towards the design of visual database querying systems, for example: QBE (Zloof, 1977), STBE (Ozsoyoglu, December 1989), AQL (Miyao, 1986), RC/S (Ozsoyoglu, September 1989), GQL/Andyne (Andyne, 1991), GDML (Czejdo, 1990), QBD* (Angelaccio, 1990), GRAQULA (IBM, 1991), GUIDE (Wong, 1982), PICASSO (Kim, 1988), and IFO (Abiteboul, 1987). However, none of these efforts utilize a dataflow approach to query specification. The dataflow paradigm provides several advantages, discussed later, that we believe form the basis of a query language that is superior to the approaches listed above. Perhaps the most important benefit of this approach is the ability of the user to treat relations as abstract entities which are operated on by relational operators. This abstraction allows the user to compose his queries strictly in the realm of relational algebra, rather than having to know the details of *how* the operations are carried out, as is the case in SQL.

C. THESIS OVERVIEW

Chapter II presents background information for the thesis. We cover the previous work done in the areas of graphical database interfaces and dataflow programming. We also expand on the motivation for a new query language to solve problems with the

current de facto standard, SQL. None of the previous work cited has been done in the area of dataflow querying. However, the different approaches of previous attempts to produce a usable graphical query interface do bring out some of the reasons for, and desired qualities of, a graphical query language.

Chapter III describes DFQL first from the conceptual point of view and then discusses the current user interface and functional details. The implementation description covers Prograph™, the object-oriented graphical dataflow programming language in which DFQL was implemented. The method of intermediate code generation and linkage to an existing backend database management system (DBMS) is discussed. The utility of programming within an object-oriented model is addressed, especially as it pertains to the extensibility and portability of our DFQL interpreter.

Chapter IV provides an analysis of the apparent advantages and disadvantages of DFQL. The results of a simple usability experiment conducted to compare DFQL and SQL are given. Also provided is background information on human factors analysis of query languages.

Chapter V provides a summary discussion of the research, and provides suggestions for future work in two general areas. The first area encompasses additions and modifications that can be made to the implementation of the DFQL interpreter program itself. The second area is theoretical investigation into extensions and optimizations of the actual DFQL language to include in-depth experimental analysis of the human factors issues of both the use and implementation of DFQL.

II. PREVIOUS WORK

A. GENERAL DISCUSSION

As stated in the introduction, we could find no previous work on a dataflow query language. In our research we have brought together the two separate ideas of a graphical query language and the dataflow programming model to create our implementation of DFQL. The following sections of this chapter discuss previously developed graphical query languages and provide a brief introduction to the dataflow programming model. In the discussions on other graphical query languages we stress the ideas behind the design and/or implementation; in most cases there is no direct comparison possible with DFQL due to the different approach. The discussion on the dataflow programming model is very brief; we assume the reader has some knowledge of the dataflow design methodology from which this model is derived.

We introduce the previous work with a section on problems that have been noted with the current de facto standard relational database retrieval language SQL. Many of the problems reviewed will be common knowledge to SQL users. We cover this topic here to expand on the motivation for the invention of our new query language; many of the criticisms raised are also applicable to the motivation for the development of some of the other graphical query languages discussed in the subsequent sections.

B. PROBLEMS WITH SQL

Our approach to exploring problems with SQL concentrates primarily on its human factors aspects. In several instances, however, items such as ease of use are influenced by more serious flaws in SQL other than simple interface design. We mitigate these problems with DFQL.

1. Basis of SQL

Query languages for the relational database model can be divided into three types: relational calculus based, relational algebra based, and a combination of the previous two types. In relational calculus, the user provides a predicate calculus expression which defines the characteristics of the tuples to be retrieved. Tuple variables are used in order to make the logical connections between separate instances of relations being combined (joined). In relational algebra the user specifies a sequence of relational operations to be performed on the tables of his schema in order to produce the desired result. Both the relational algebra and the relational calculus have equal expressive power (Elmasri, 1990, p. 212); any query that can be expressed in one can be transformed into a query in the other (Codd, 1972).

SQL is a mixture of both relational algebra and calculus with some other ideas (such as nesting to provide a block structure) thrown in also. However, SQL tends more toward a calculus based approach because it is primarily a declarative rather than a procedural language. The user specifies what the result should be in one statement, rather than in a sequence of statements. (Dadashzadeh, 1989, p. 431) This mixture of approaches resulted from the way in which SQL was designed. Date comments:

> When the language [SQL] was first designed, it was specifically intended to differ from the relational calculus (and, I believe, from the relational algebra). ...As time went by, however, it turned out that certain algebraic and calculus features were necessary after all, and the language grew to accommodate them. (Date, 1987, p. 84)

The result of this design methodology is that SQL is a strict implementation of neither relational algebra nor calculus.

2. Declarative Nature

SQL is primarily a declarative query language. That is to say that the user is intended to construct his query basically from the relational calculus or first-order predicate logic frame of reference. All of the conditions that the query result is desired to meet are specified in a single statement. This approach is straightforward for simple queries; for more complex queries however, the logical expression specifying the conditions to be met can become quite complicated. This problem is exacerbated when the complex query involves universal quantification (discussed separately below) because of its negative logic implementation in SQL.

Another problem with the declarative approach is that it may not always present the clearest representation of the query to the user. The question of clear representation of the "essence" of the query is in part related to how humans actually think. The logical, declarative method of expression may be more compact, however, if humans think of complex problems in a more sequential fashion it may not be as easy to formulate or to interpret after formulation. Indeed, in *The Relational Model for Database Management: Version 2*, Codd uses the algebraic (procedural) approach to introduce the operators for the relational model because "upon first encounter, that approach appears

easier to understand." He goes on to say that the relational calculus provides a better implementation for an actual database language not because of any of its ease of use characteristics for the humans but because it is easier for the computer to optimize a query that is confined to a single condensed command. (Codd, 1990, p. 62) Most embedded query languages, and even some commercial implementations of non-embedded query languages, give the user the ability to use the query language in a procedural manner if desired. This allows the user to take advantage of the features of the host language to accomplish operations that are difficult or impossible to code in the query language.

Ease of use issues for database query languages have been of concern for quite some time as evidenced by Schneiderman's paper "Improving the Human Factors Aspect of Database Interactions" published in 1978 (Schneiderman, 1978). Human factors studies have been done concerning the issue of declarative versus procedural implementations of query languages. The results of these studies show that, for complex queries, users perform better when using a procedurally oriented rather than a declarative language such as SQL. (Welty, 1981)

3. Universal Quantification

The idea of universal quantification is expressed in English by the phrase "for all." This idea is often required to formulate database queries but is supported only indirectly in SQL. One of the problems of using universal (or existential -- "there exists") quantification is that the logical meaning of these operations is not completely intuitive, especially to persons who are not experienced in the use of predicate logic. The logical

ideas represented by these operators have been shown to be difficult to use correctly even when the user has experience in the area in which they are being applied. (Negri, 1989, p. 90) This difficulty is compounded because SQL's lack of a specific "for all" operator forces one to use "negative logic" with the existential quantifier (NOT EXISTS) to achieve the result of universal quantification.

The following example is provided to show how, even with a simple query, expressing the idea of universal quantification is somewhat complicated. As the query becomes more complex, the difficulty of composing or understanding it increases rapidly.

Consider a database with the following relations (key attributes are underlined):

COURSE(CNO, TITLE, INO)

ENROLL(CNO, SNO, GRADE, TESTSCORE)

INSTRUCTOR(INO, INAME, PAY)

The desired query is: "list the names of all the instructors who gave all A's in at least one of the courses they taught." (This database schema is a subset of our example database schema described in Appendix A.)

In SQL the above query could be expressed as:

```
(1)     SELECT DISTINCT INAME
(2)     FROM INSTRUCTOR, COURSE, ENROLL E1
(3)     WHERE INSTRUCTOR.INO = COURSE.INO
(4)     AND COURSE.CNO = E1.CNO
(5)         AND NOT EXISTS
(6)             (SELECT *
(7)              FROM ENROLL E2
(8)              WHERE E2.CNO = E1.CNO
(9)                 AND E2.GRADE != 'A');
```

A direct english translation of the SQL query above is: "Retrieve the names of instructors (line 1) who taught courses (line 3) which had students enrolled (line 4) in which there was at least one of these courses in which there was not any student who received a grade that was not an 'A' (lines 5-9)." This translation describes only the basic semantics of the SQL statement. In order to derive the full meaning of the SQL query and how it will function, a knowledge of the differences between relation names and range variables and their scoping rules is required. This example query would not be possible without the correct use of range variables; the linkage between the "inner" and "outer" SELECT statements depends entirely on understanding and correctly employing range variables to represent the ENROLL relation. The specification required to form this simple query is thus not straightforward at all; even at this fairly low level of complexity, the query formulation involves subtleties of logic that are extremely easy to mixup, even for the experienced user. A final comment on this SQL example is on its readability: How difficult is it to read the SQL query and know what is actually being specified? If one has to intensely study a previously written query to determine exactly what it is going to do, the implication is that there is a comprehension problem caused by the language. Lack of easy comprehension of the language will affect not only query readability but also the capability of the user to formulate correct queries.

4. Lack of Orthogonality

"Orthogonality in a programming language means that there is a relatively small set of primitives that can be combined in a relatively small number of ways to build the control and data structures of the language." (Sebesta, 1989, p. 6) SQL does not

present the user with a simple, clean, and consistent structure. In SQL there are numerous examples of "arbitrary restrictions, exceptions, and special rules." (Date, 1987, p. 84) An example of an unorthogonal construct in SQL is allowing only a single DISTINCT keyword in a SELECT statement even if the SELECT statement contains other nested SELECT's. Lack of orthogonality in a language increases the number of special rules that must be memorized by the user, decreases the readability and writability of the language, and in general decreases the usability of the language. Unorthogonal features are not something that we have to live with to have a powerful query language, this fact is evidenced by the orthogonality of DFQL.

5. Nesting Construct

SQL permits a nesting structure of the form--[1]

```
SELECT
FROM
WHERE attribute IN
    SELECT ...
```

This format allows for a block structure type of construct. In fact, it is from this construction that the "Structured" in "Structured Query Language" is derived from. The original purpose of this nesting construct was to allow the specification of certain types of queries without resorting to the use of relational algebra or relational calculus. However, with the introduction of these same relational algebra and calculus operations into SQL the need for the "IN subquery" construct was eliminated. (Date, 1987, p. 84)

---

[1]There are also other forms of nesting allowed.

Codd refers to the nesting construct as part of the "psychological mixup" in SQL. While all queries that are specified using the nesting construct should be directly translatable into queries using an equi-join instead, Codd shows that if allowing for the existence of duplicate rows in tables (as SQL does), one will come up with a different result when performing the equi-join version of the query than when performing the nested version (Codd, 1990, p. 380). There is also a problem in the optimization of nested queries by the DBMS. Although work has been done to demonstrate the translatability of nested queries into their non-nested counterparts (Kim, 1982), most optimizers perform poorly, if at all, in optimizing through levels of nesting in SQL queries. Thus, a performance related issue is thrust onto the user simply through the design of the query language. (Codd, 1990, pp. 379-382)

6. Lack of Functional Notation

The use of functions in programming languages allows the abstraction of operational detail to whatever level is appropriate for the environment in which the function will be executed. In the same fashion, complex queries that provide an intermediate result that is used for a higher level query could effectively be hidden from the user at the higher level through the use of functional notation. This concept is used in all modern programming languages, even including newer versions of BASIC, but is not implemented in SQL. The use of functions to produce intermediate results for further processing also provides a good alternative to query nesting by allowing the hiding of complex query structures from the user. User-defined functions are discussed by Codd as a desired part of the relational model (Codd, 1990, p. 340-344).

7. Other Issues

There are other various problems with the SQL database language, however, most of these are not germane (or at most peripheral) to our thesis on DFQL. Many of these arguments are not necessarily with the "language" portion of SQL but in its actual DBMS implementation of the relational model. Examples of these types of arguments are: lack of adequate support for keying and referential integrity (Date, 1987, p. 84), lack of support for three or four valued logic (Codd, 1990, pp. 383-386), and allowance of duplicate rows in tables (Codd, 1990, pp. 372-379). Our research concentrates on the *interface* to the underlying relational DBMS. We assume the underlying DBMS supports the relational model as defined by Codd (Codd, 1990).

C. EXISTING VISUAL QUERY LANGUAGES

In our overview of previous research efforts into the area of visual database interfaces, we have found that there have been two basic directions pursued. The first direction is a forms based interface to the relational model as exemplified by QBE (Zloof, 1977). The second direction uses the entity-relationship model's representation of database objects as the basis on which the user can construct his queries. We will group our discussion into these two areas, citing particular efforts in each, and list the objectives, benefits, and drawbacks of each approach. An additional section is also provided which discusses interfaces that do not fall into either of the two preceding categories.

1. Forms Based Interfaces

The ancestor of all of the forms based interfaces is QBE (Query by Example), developed at IBM circa 1976 (Zloof, 1977). The basic idea behind QBE is that the user calls up a form which represents the attributes in a given table. This idea should be familiar to anyone who has used a spreadsheet type program. The user types *example values* into columns which represent attributes in a specified relational table, and the DBMS then returns the tuples that match the example values that the user entered.

Figure 1 shows an example query in QBE based on the example database specified in Appendix A. The english translation of this query is: "Retrieve the names of the instructors who gave at least one 'F' in any of their classes."

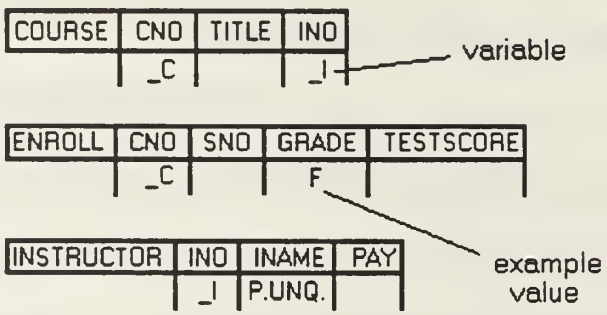


Figure 1. Example of QBE Query

QBE uses variables to specify queries, in the spirit of domain relational calculus. In the example above, the variables "_C" and "_I" are used to join the three tables required to answer the query. They are *free domain variables* which can represent any value in the domain of their specific column. By using the same variable name in more than one table, the user specifies a join on that column. Conditions to be satisfied

by the query are also entered into the column they reference. Hence, in our example we place an "F" in the GRADE column of the enroll table to specify that we only want to retrieve rows where the GRADE attribute was equal to "F". Other relationships can also be specified such as >, <, !=, etc; equality is the default relationship. For complex relationships it would be unwieldy to specify the condition by writing it in the actual column box. A separate box, called the condition box, is called up for this purpose. "P." specifies columns to print out as the result of the query. "P.UNQ." means to print out only unique values of the specified column, much like the "SELECT DISTINCT ..." statement in SQL. Thus, in our example, we will produce as output a list of INAME's meeting our desired condition with each name only appearing once.

Although QBE is very nice for allowing relatively inexperienced users to specify simple queries, it becomes less and less useful as query complexity grows. Expressing universal quantification in QBE is difficult. While there was provision for expressing universal quantification in the proposal for QBE, it has not been implemented. In fact IBM's QMF (Query Management Facility as implemented under DB2), which was used as the basis for our example, provides no support for existential or universal quantification and thus is not relationally complete; universal quantification *cannot* be specified (Elmasri, 1990, pp. 241-249). This means that the example query we presented in section A.3 could not be expressed using QBE.

QBE is one of the first query languages to support a *two-dimensional syntax.* This places it among the earliest "visual" database interfaces, where here "visual" means not purely textual. Use of a form template in which to express the query was viewed as

a natural interface for people in offices who were accustomed to dealing with fill in the blank type forms (Shu, 1988, pp. 239-240). The design of the QBE interface was also directly influenced by the type of hardware available at the time of development. In the late 1970's, bit-mapped graphics terminals were not widely available, so the developed interface had to run on standard character based terminals. This limits the format of information displayed to tables, as shown in our example.

The success of QBE in providing a user-friendly interface for relatively simple ad hoc queries has led to a number of other database query languages being developed along the forms based design. We briefly discuss several of these languages in the following sections.

a. STBE--Summary Table By Example

STBE (Ozsoyoglu, December 1990) has been developed to solve problems unique to the area of statistical database applications. The primary motivation is the repetitive production and comparison of summary tables in these applications. The basic query facility provided is very similar to that of QBE. The extensions provided allow STBE to deal with relations that have set-valued attributes, summary tables, and aggregate functions using queries that by nature have a hierarchical subquery structure. STBE is based heavily on set and aggregation operations because of the type of data manipulations that are expected to be performed. While there is no explicit implementation of universal quantification, STBE uses set comparison operators (which may be nested) to achieve the same effect. STBE could be considered relationally complete since it can implement Cartesian product, projection, selection, set union, and set difference; however, STBE

departs from the pure relational model by allowing summary tables and relations with set-valued attributes.

The graphical interface for STBE is much like that of QBE in its format. It uses skeleton tables that are filled in with variables and conditions to compose the query. STBE introduces the idea of scoping by allowing nested queries. A nested query is implemented by placement of the table skeletons for the query in nested windows. The variables in each table skeleton in a given window are bound throughout that window. In a nested query, each window contains a subquery and behaves somewhat like a function returning an output. This output is specified by either an *output relation skeleton* or an *output summary table skeleton* in the owning window. The outermost window is always named ROOT; the ROOT window's output is returned as the response to the query. This structure of nested windows leads directly to a digraph representation of the query which can be formed by decomposition of the STBE query into a parse tree. The condition box in STBE performs the same function as in QBE but also allows set membership ($\in$, $\notin$) and set comparison ($\subset$, $\supseteq$, etc.) along with the normally supported relational (<, >, =, etc.) and boolean (AND, OR) operators.

The powerful aggregation features, handling of summary tables and relations with relational attributes, and nesting structures make STBE "not very simple as a language" as admitted by its designers. This language is intended for the specific field of Statistical Database Management where it would be used by advanced users and not novices. (Ozsoyoglu, 1989, p. 566)

b. AQL--A Query Language

AQL (Miyao, 1986) is implemented for the AIDE-II (An Intelligent Database System for End Users) prototype database management system. AQL is another two dimensional query language that is extremely similar to QBE. The major difference between the two is that in AQL there is no need to express joins between tables. This is due to the design of the AIDE-II data model in which a "user view" is specified that is supposed to contain all of the possible relationships in the database. The expressive power of AQL is contained in that of QBE. (Miyao, 1986, p. 27)

The lack of the ability to express joins and universal quantification are serious drawbacks of AQL. The elimination of the join operation from the query simply splits the query up into two dissimilar parts: first specifying the user view required for the query, and then actually specifying the conditions for the query on that user view. AQL's inability to support the relational model is a fatal drawback.

c. RC/S--Relational Calculus/Sets

RC/S (Ozsoyoglu, September 1989) is a relational calculus which uses set comparison and set manipulation operators to replace universal quantification in query formulation. Two graphical implementations of RC/S have been designed, both of which are heavily based on QBE. The first implementation uses nested windows to specify complex queries, as discussed for STBE above (however RC/S is for a simple relational database). The second implementation provides the same functionality as the first, but uses hierarchical windows to express the nesting concept. RC/S was developed by the

same principal as STBE; the query constructs are nearly identical with the exception of RC/S handling only simple relations.

d. Objectives, Benefits, and Drawbacks

The initial objective of forms based interfaces was to provide the user with a way to construct queries based on objects that he was familiar with, namely forms (Shu, 1988, p. 239). QBE was the first implementation of a two dimensional query language and for simple queries seems to be "easy to use." However, QBE (as implemented in QMF) is not even relationally complete and therefore cannot express some types of queries that a user may desire (for example, queries involving universal quantification). STBE and RC/S (graphical) attempt to mitigate this problem while retaining the ease of use characteristics of QBE. We believe that they are only partially successful. The implemented nesting and the use of set comparison do allow the expression of the categories of queries that are not expressible in QBE; however, these same added features detract greatly from the simplicity of the language. The correct use of set operations to solve the universal quantification problem requires at least some knowledge of set theory. This is an additional burden placed on the user on top of learning the semantics of the query language itself. AQL eliminates the user specified join from the actual query by requiring a "user view" schema to be set up prior to the execution of the query. We believe that this unnecessarily separates the query building process into schema manipulation (the creation of the user view) followed by actual query specification and is not an aid to the user. Also the AIDE-II DBMS for which AQL is

designed falls outside of the definition of the relational model due to its requirement of user views.

2. Entity-Relationship Model Interface

The entity-relationship (ER) model was introduced in (Chen, 1976). The ER approach has been used extensively as a high-level conceptual data model. The idea behind the model is to illustrate the concepts of entities and relationships between entities in a graphical way in order to enhance understanding of the structure desired for a database. In the past, the ER model was used as an aid in developing the structure of the database that would then be implemented using a relational DBMS and its associated query language, but recently several query languages have surfaced which are based closely on the ER model.

The normal visual representation of the ER model is shown in Figure 2 (using the example database described in Appendix A). The rectangles represent entities and the diamonds represent relationships between entities. Both entities and relationships may have attributes which are represented by the connected ovals. This representation is intended to specify some of the semantics which are contained in the database.

One of the drawbacks to the ER approach is that just because certain relationships are currently specified does not necessarily mean that there are no other relationships that exist between entities. When this type of representation is used as the basis for a query language, it tends to force the user to depend on the specified relationships. Indeed, the idea for using the ER diagrams is that the user need not worry about the specific "join" conditions between entities. These "relationships" are all

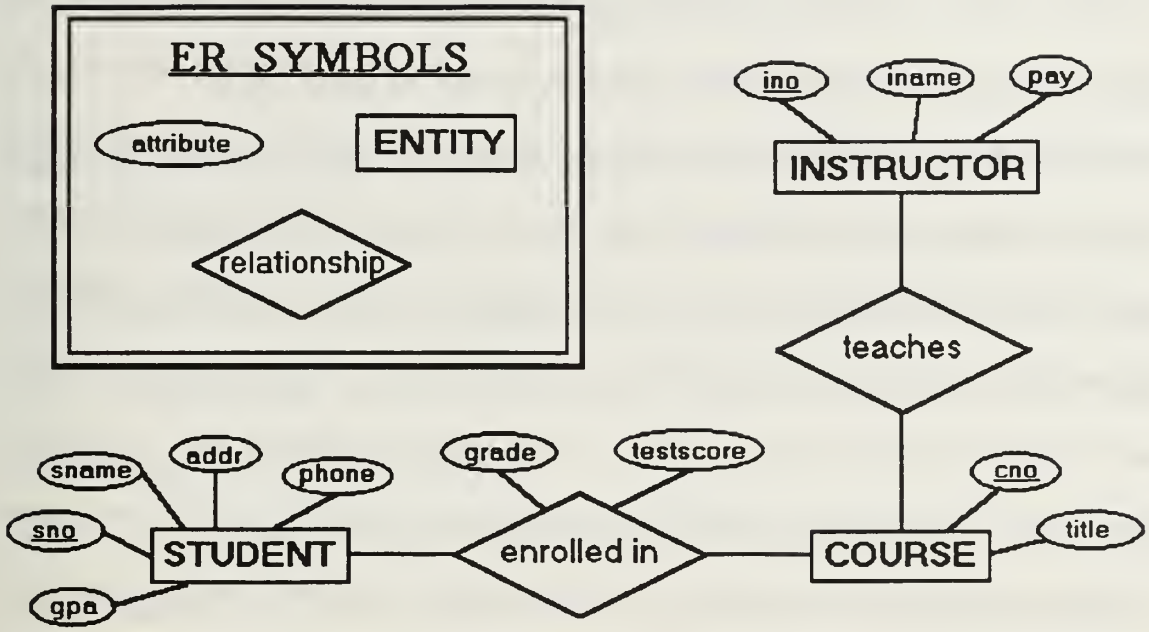


Figure 2. Example of ER Diagram

displayed for him. (Although *someone* at some time has to define these relationships.) This is similar to the idea in AQL where user views are specified so that all joins are eliminated from the user's purview. Dependence on predefined relationships may provide benefits to the novice user who doesn't really understand how the data in the database fits together; but it seems somewhat dangerous to write queries which depend on relationships that the user may not fully understand. The ability to easily use a relationship without knowing how it is actually set up increases the chance of syntactically correct queries that will produce the wrong results.

The ER query languages also present a severe restriction on the advanced user if they do not allow relationships other than those previously specified in the ER schema. This is the case in several of the ER query language products. Since most relationships

in the ER model are based on equi-joins on keys and foreign keys of the entities, the restriction may not seem too onerous; indeed, it may appear that if the ER schema is set up correctly, there could be no relationships that are left out. However, if the user desires a theta-join based on some relationship other than equality, even if this theta-join uses the same key attributes as one of the defined relationships, it would be impossible to perform without adding it as a "new" relationship to the ER schema. In any case, the ER type query languages require the formulation of the query to be divided into two distinct phases. First, the appropriate relationship (or relationships) must be found in the schema (or created). Second, the actual query conditions must be specified.[2] Requiring the user to perform two dissimilar steps in order to construct a query does not allow him to maintain a smooth train of thought while formulating the query.

Most of the graphical query language implementations based on the ER model are designed around providing the user ways in which to manipulate the displayed schema in order to specify his query. The idea has some intuitive relation to QBE; instead of placing conditions on forms representing the schema, one places the conditions on the ER diagram. The actual method of graphical implementation of the ER diagrams seems to depend primarily on the type of hardware that the implementation was designed on or intended to run on. Thus, the interface types run the gamut from Macintosh point-and-click to more simplistic line drawings. We briefly discuss some ER type query languages in the following sections.

---

[2]This is the same as the problem caused by user views in AQL (p. 18).

a. GQL/Andyne--Graphical Query Language

GQL (Andyne, 1991) is a commercial product developed by Andyne Computing Limited of Kingston, Ontario, Canada. It is designed to run as a front end to a user's existing relational DBMS. GQL runs on Macintosh computers and thus the Apple "look and feel" is very much a part of the GQL query interface. In this discussion, unless otherwise stated, we are referring to the product called GQL/User which is the Andyne's user query language interface.

On startup, GQL displays the appropriate ER diagram for the database that the user desires to run his queries on. Also provided on the startup screen may be several "executive buttons" which are used to run previously stored or "canned" queries that may have been written by or for the user. To perform single table (or perhaps more appropriately single entity) queries, the user double-clicks on the icon in the ER diagram representing the desired entity. A window with a list of the entity's attributes is then displayed. Attributes may then be selected for printing. A "filter" dialog[3] is provided to aid the user in formulating requests to sort the data or actually "filter" it by only including items meeting certain criteria: minimum, maximum, or value ranges may be specified. Queries for specific items, such as "list the name and address of student with sno = S123 or sno = S321", are formulated with the assistance of GQL's "qualify" feature. To "qualify" an attribute, a dialog is displayed where the user enters the condition to satisfy; if previous conditions exist, the user must also select whether to

[3]For consistency we use the Apple spelling of "dialog" throughout this thesis.

"and" or "or" the new condition with the previous ones. The user is stepped through all of the qualification steps by GQL.

The information represented by the relationships in GQL is accessed by selecting the desired relationship from the screen along with its two (all relationships are binary) adjoining entities. Now, when a query is formed, all of the attributes from both entities are available for qualification and display. The relationships that the user sees must be entered by the database administrator (DBA). These relationships are neither changeable or extensible by the user. Another problem, due to the representation of the ER model, is that for a complex database with many relationships the diagram will be too large to fit on the screen at any one time. This may be even more likely to happen with GQL than other ER products because GQL is designed to be run on top of an existing relational DBMS. Each of the tables in the existing database becomes an entity. The DBA must then define a relationship for each of the possible join conditions in the database. If the underlying database has many tables (which is especially true of large database schemas that have been reduced to third normal form (3NF)) with many join conditions the resulting ER diagram will necessarily be very large. The ER diagram will not be easy to use if it does not all fit on the screen at the same time.

b. GDML--Graphical Data Manipulation Language

GDML (Czejdo, 1990) uses much of the same type of pictoral representation as the general ER model and GQL/Andyne. This query language is based on an extended version of the ER model that incorporates "...various forms of generalization and specialization, including subset, union, and partition relationships."

(Czejdo, 1990, p. 26) Queries are formed in GDML by removing parts of the ER diagram. An editor is provided to allow the user to erase parts of the ER diagram. All of the items in the database represented by the diagram remaining on the screen are then displayed as the result of the query. A method of restriction is provided by allowing the user to place conditions on the attributes in the diagram. Although GDML is based on the ER model for the user interface, as implemented it runs on top of a relational DBMS. The GDML entities are simply relations from the underlying database and the GDML relationships are represented by database relations containing the appropriate keys from each of the connected entities. Again, as with GQL, these relationships must be established manually.

As an example, we will use the schema from Figure 2 to solve the query: "Retrieve the names of students who received one or more 'A' and also the name of the course they received it in."[4] First, remove the INSTRUCTOR entity from the diagram. Removing an entity will also remove all attributes and relationships tied to it. Thus, the "teaches" relationship is also removed. Then remove all of the attributes from the diagram with the exception of the ones to print, namely "title" and "sname". Next, use the *restrict* operator to add the condition "grade = 'A'" to the relationship "enrolled in". The construction of the query is now completed. The results are produced by selecting the *display* operator. When *display* is selected all of the necessary joins are performed and the tuples from the resulting relation are displayed for the user.

---

[4]We are intentionally using different queries for our examples in order to best display the unique characteristics of the particular languages.

c. QBD--Query By Diagram**

QBD* (Angelaccio, 1990) is intended to be a "user-friendly" query language based on the ER model which allows the expression of queries with a recursive nature. QBD* uses the ER diagram as a navigational tool for forming queries. The actual conditions to be satisfied by the query are specified in separate query specification windows.

To use QBD*, the user first selects items of interest from the displayed ER diagram. When an item is selected, a window is opened to allow the user to place conditions, including recursive conditions, on the attributes of that item. The conditions are built up by the use of icons representing the standard comparison operations such as >, <, =, for example. The query condition window displays the attributes of the selected item in a column on one side of the screen. To compare a given attribute to a value, the value is entered on the opposite side of the screen and a line is then drawn between the attribute and the specific value. A comparison operator is then selected from the icon list at the top of the screen and is attached to the line connecting the attribute and the value. By placing two separate entities on either side of the screen, join conditions can be specified between two separate relations. By duplicating the same entity on both sides of the screen recursive queries may be specified. (Angelaccio, 1990, p. 1154)

The two types of windows that are used in QBD* are quite different from each other. This is because they serve entirely different purposes, however the dissimilarity makes the linkage between the ER diagram and the actual query specification seem somewhat tenuous. The two types of windows are used to accommodate the

designers' choice to implement the query formulation process as a series of phases: First the user browses the schema. Then he picks the required items (or concepts as QBD* calls them) from the ER diagram. Next, the selected sub-schema is transformed to "bring it 'close to the query'" (Angelaccio, 1990, p. 1152). Finally, the "navigation phase" is entered where the actual query is formed in the query condition windows. This series of steps seems unnecessarily complex. The formulation of the query in the query condition windows also allows the user many options which are not based on the relationships specified in the given ER model. For example, QBD* allows the specification of joins (relationships) which are not reflected in the ER schema being used. If a query system is to be based on the ER model, then the implementation should stay within the bounds of that model. If these joins are truly necessary, then they should be reflected as part of the ER model, according to the philosophy of that model. This anomaly arises from an attempt to provide flexibility that is missing from the underlying ER model.

d. GUIDE--Graphical User Interface for Database Exploration

GUIDE (Wong, 1982) has been developed especially to allow the browsing of metadata in large databases with many complex relationships. Its design and display methodology is based on the ER model, but GUIDE allows the user to select a level of detail with which to look at the database. To handle metadata, entities are organized into a "hierarchical subject directory" and attributes are organized into a "hierarchical attribute directory." The purpose of these directories is to guide the user to the part of the ER schema that is relevant to him. Also, a facility is provided to "rank" objects according to their expected relevancy to a certain group of users. This ranking

is based on the objects expected "importance" in the system. The ranking does not necessarily correspond to the hierarchical organization discussed above, but should reflect the interests of the group of users and the frequency of access to that object by them.

To formulate a query, GUIDE asks the user to first select the level of detail to display for the schema. The ER diagram is then presented at the desired level of detail. Indirect relationships between entities (the actual connections are not shown because they involve objects at a lower level of detail) are represented by dotted lines between entities. Next, the attributes of the displayed entities and relationships can be examined by selecting the desired object and then "examining" that selected node. Examining a node will again present the user with a hierarchical description of the attributes of that node; information is also provided on what that attribute represents and what values or codes are allowable for the attribute's data. Restrictions can be placed on selected attributes in order to specify the query. The user may select separate portions of the schema to run partial queries, while still maintaining any previous queries. These separate partial queries may then be combined to form a final query.

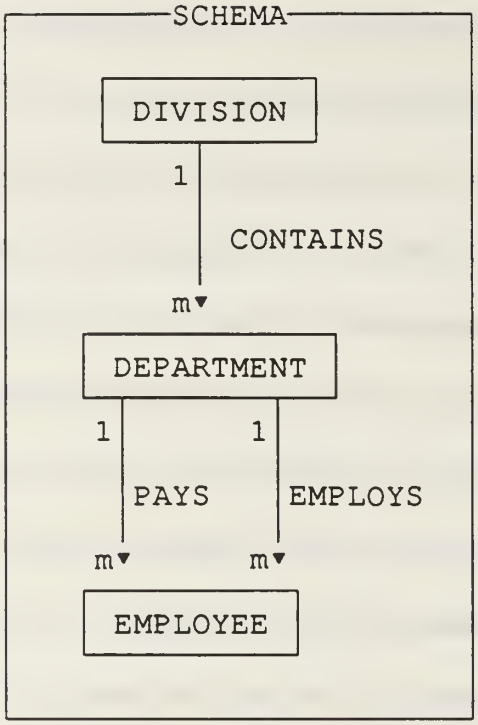
e. GRAQULA--Graphical Query Language

GRAQULA (IBM, 1991) is being developed by IBM as a graphical language for querying and updating a database. GRAQULA has both an ER and a relational implementation, but we have placed it in the ER category of query languages because the basis of GRAQULA is more related to the ER model. The syntax of both the ER and relational versions of GRAQULA is similar; the relational version depends on the specification of referential integrity constraints and, optionally, expected joins to

provide the connections between relations that would be given by the relationships in the ER model. We will discuss only the ER version here.

GRAQULA is based on the definition of a database schema that is presented to the user in the form of an ER diagram. The relationships are displayed simply as directed arcs between the entities with the appropriate relationship name attached to the arc. The database schema is displayed in one window while the query is built up in a separate query window (as shown in Figure 3). The query window is initially empty. The user selects entities from the schema window; they are then displayed in the query window for further manipulation. To formulate the query on the items the user has placed in the query window, the items may be expanded to show their attributes. The attributes are listed in a tabular fashion and restriction conditions can be entered for them somewhat as in QBE. Joins between items which are unrelated in the schema can be performed by specifying the join attribute from one entity in the other entity's value column.

In Figure 3 some of similarities between GRAQULA and QBE are apparent. The query represented by this figure is: "List the name and salary of each employee whose salary exceeds 50,000 and whose year of hiring equals the Research division's year of formation." (IBM, 1991, p. 13) This query requires a join between the EMPLOYEE entity and the DIVISION entity on the YEAR_HIRED and YEAR_FORMED attributes. The join is represented by including DIVISION.YEAR_FORMED as a comparand for YEAR_HIRED in the EMPLOYEE entity. If any relationship had been previously specified between the EMPLOYEE and



(IBM, 1991, p. 7)

QUERY

-		EMPLOYEE	
ATTRIBUTE	OP	VALUE	(AND)
NAME **SALARY** YEAR_HIRED	>	50000 DIVISION.YEAR_FORMED	

-		DIVISION	
ATTRIBUTE	OP	VALUE	(AND)
NAME BUDGET YEAR_FORMED		'RESEARCH'	

(IBM, 1991, p. 13)

Figure 3. Example Join in GRAQULA

DIVISION entities, there would be a line drawn by the system between the two entities with the name of the relationship on it. In GRAQULA, conditions are specified by filling in the VALUE column of the displayed entities. The relational operator (OP column) is assumed to be equality ("=") if it is left blank. In this case, the conditions are all conjunctive, as indicated by the "(AND)" in the right corner of the value column. As in QBE, complex conditions can be formed in a condition box that is then attached to the query.

Additional power is added to GRAQULA by nesting simple entities and relationships inside various frames. A frame is indicated by a box drawn on the screen which may contain one or more entities and their associated conditions and relationships. These frames are used to specify logical operations such as simple conjunction and disjunction (AND and OR), negation with conjunction and disjunction (NAND and NOR), and implication and consequent. The logical operations are scoped over any of the entities and relationships that are contained in their frame. Nesting of operations can thus be performed by nesting frames, providing a clear way of showing the scope of each of the operations. The inclusion of *implication and consequent* frames is intended to ease the problem of specifying universal and existential quantification. As stated previously, the predicate logic approach for these ideas is not simple. The implementation of the implication and consequent ideas does not do much to simplify the process of quantification specification because of the complex nature of the idea involved. Sockut proposes a method for transforming quantification queries from English into GRAQULA

statements (IBM, 1991, p. 23). This procedure is non-trivial and the meaning of the resulting GRAQULA query is not obvious.

f. Objectives, Benefits, and Drawbacks

The primary objective in the proposal of the ER model based query languages is simplification of the query specification process for the end user. A significant benefit of the ER query approach is that the database schema is displayed so that the user does not have to memorize the specific relationships between database objects. However, this is also a drawback. Using the actual schema to define queries limits the user to the predefined relationships that have been coded into the schema. Even in systems that allow the user to define his own relationships, the user is forced to break up a single logical query into two disjoint and dissimilar steps.[5]

Most ER systems assume relationships based on the equi-join of keys between entities. This does not take into consideration relationships based on other attributes or on other types of theta-joins. In systems that do allow the user to perform joins without having them defined in the ER schema, GRAQULA for example, convenience is added at the expense of violating the ER model. If the user is joining entities based on certain attributes and conditions, then this *relationship* should be indicated in the ER diagram. Without enforcing this rule, the semantics of the database schema and its associated ER diagram are lost by a buildup of stored queries based on

---

[5]p. 22

specified joins. The actual relationships that are being used may never make their way into the database ER diagram; semantic correctness of the model is lost.

Another problem with the ER model in general, is that the distinction between entities and relationships in the schema is not necessarily straightforward: "...one person's entity is another person's relationship." (Codd, 1990, p. 477) An example of this ambiguity is presented in the representation of an airline flight. To an accountant it exists as an entity--a concrete object. To a scheduler it exists as a relationship between a specific aircraft, aircrew, routing, date, etc. (Codd, 1990, pp. 477-478) Neither of these determinations are wrong, they are just based on the different points of view of the people involved. However, this lack of concrete distinction could cause problems when queries must be made from a single ER schema by multiple users, each with a different point of view.

3. Other Approaches

Although most of the graphical query languages proposed fall into one of the previous two categories (forms based or ER model based), we will briefly discuss two approaches that differ somewhat from either of these two previously discussed categories.

a. PICASSO--Picture Aided Sophisticated Sketch Of Database Queries

PICASSO (Kim, 1988) is a graphical query language that is structured heavily on the universal relation database model. The idea behind a universal relation database is that all of the join dependencies are included in the universal relation itself.

This relieves the user from the necessity of knowing which relations database attributes are attached to, since there is only one relation--the *universal relation*.

PICASSO uses hypergraphs to represent the semantics of the database. Attributes of the universal relation become nodes in the hypergraph. Hyperedges are formed by collecting the attributes that have *fundamental relationships*; thus, the hyperedges form conceptual objects. A second hypergraph is then constructed with the conceptual objects as nodes and the hyperedges representing *maximal objects*, or the maximal sets of objects in which queries "make sense." (Kim, 1988, p. 172) Attributes that are in common between two (or more) maximal objects are shown by having those parts of the hypergraph overlap. Figure 4 depicts an example PICASSO hypergraph. This PICASSO example contains information only on courses and instructors from the schema of Appendix A.

To form a query based on this hypergraph, the user would use a pointing device to place a question mark next to the attributes that he would like to be returned from the query. Simple selection can be performed by attaching selection conditions to attributes on the screen. PICASSO allows selection conditions using not only the normal relational comparison operators (<, >, =, etc.) but also grouping and set operators. An example query based on Figure 4 would be constructed by appending " = 'SMITH'" to INAME. If the TITLE attribute had a question mark next to it, the query thus formed would produce a listing of the course titles that 'SMITH' taught. This is a very simple example of how a query is formed in PICASSO. There are many other rules for forming more complex queries.

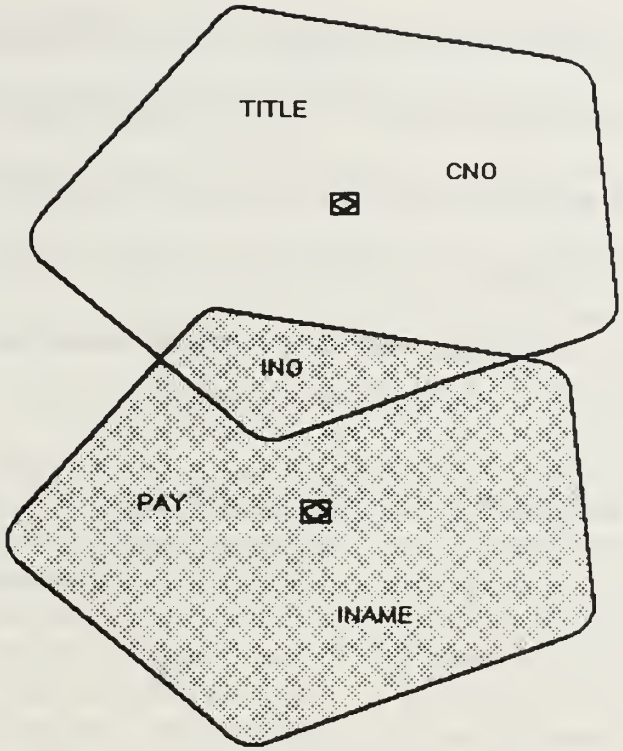


Figure 4. Example PICASSO Hypergraph

One of the major drawbacks of PICASSO is the limited amount of information that can be displayed on the screen at any one time. The hypergraph drawing for even our simple example is rather large, and most actual databases would have schemas much more complicated than two maximal objects. Also, when several related hypergraphs are displayed on the screen simultaneously, the picture rapidly becomes confusing.

The expression of joins in PICASSO is made possible by allowing the user to create copies of a selected hypergraph and then relate the attributes from one copy of the hypergraph to the other. Again, this representation does work, but the idea of the universal relation database model is that this type of query is abnormal; performing a join

actually violates the universal relation paradigm. In fact, the designers of PICASSO admit that their graphical representation is not well suited for some complex types of queries. Their solution to this shortcoming is the use of a textual/windowing tool called "ANSWERTOOL" in which partial queries can be processed (Kim, 1988, pp. 189, 191). A single interface to the database is superior if it can be demonstrated that it is easy to use for all types of queries.

b. IFO and SNAP--A Graphics-based Schema Manager

The IFO model (Abiteboul, 1987) is an actual incarnation of a semantic database model. The ideas embodied in IFO have much in common with the precepts involved in object-oriented approaches to data modeling. Various types of atomic and composite objects are specified by the IFO model. Aggregation and ISA relationships are directly represented. Relationships between objects are specified in a functional manner. One end of the relationship serves explicitly as the domain and the other end as the range of the function. This specification supports the hierarchical construction of *fragments*. Fragments allow portions of the schema to be condensed (inside the fragment) in order to provide a modular view of the schema. This feature is somewhat similar to GUIDE's ability to provide views of the database schema at various levels of abstraction.[6] However, the IFO model is much more complex than the ER model.

The SNAP system (Bryce, 1986) is the interactive, graphical interface to the schemas of the IFO model. SNAP is primarily configured for the creation and

---

[6]pp. 27-28

maintenance of IFO schemas, however, a limited query facility is also provided. This query facility permits the expression of only selection-type queries (Bryce, 1986, p. 156). SNAP presents a screen display of the IFO schema containing the IFO objects and their connections. To place a query, the condition to be satisfied is entered in the corresponding object box. The version of SNAP discussed here only supports simple conditions; logical conjunction, disjunction, and negation are not supported. The idea of joins can be expressed in SNAP by using *comparitor arcs* to specify a comparison (<, >, =, etc.) between objects. Set comparisons can also be specified with comparitor arcs. The information to be printed upon execution of the query is indicated by highlighting the desired objects with the pointing device. If the object highlighted is an abstract, non-printable type, the appropriate printable key value for instances of the object meeting the query criteria are printed. (In IFO it is required that each object have a unique printable attribute to be used as a key.)

Due to its limited capabilities, SNAP is not complete as a query language. The data model aside, SNAP's provided query facility is similar to several of those proposed for use with the ER model and as such has many of the same types of strengths and weaknesses as those languages. However, the IFO model is much more complicated than the ER model, adding an additional level of difficulty in formulating IFO queries which is not mitigated by the SNAP system.

D. DATAFLOW PROGRAMMING LANGUAGES

Dataflow diagrams have been used in computer science as an aid in systems analysis and systems design for about 15 years. The same methodology has been used by operations research scientists for nearly 70 years. The idea behind the dataflow diagrams is to provide an easy to understand way of describing a network of functional processes which are interacting with each other based on the flow of data from one process to another. (Yourdon, 1989, pp. 139-140) Dataflow programming languages take the ideas specified by the dataflow diagram and make them directly executable. In other words, rather than using the dataflow diagram as a tool in designing a computer program, the diagram becomes the program itself.

1. Dataflow Diagram Description

A traditional dataflow diagram makes use of several distinct graphical symbols to convey its meaning. Dataflow diagrams have a strong tie to the depiction of directed graphs. In a dataflow diagram, the processes, data stores, and terminators are the nodes and the dataflows are the arcs of the directed graph. A circle is used to represent a process that is performed on data.[7] Arrows indicate data flowing from one node (most often a process) to another. These arrows are labeled with the name of the data that they represent. A square represents a terminator, an entity external to the system being modeled. (Yourdon, 1989, pp. 141-149) Figure 5 is an example of a simple dataflow diagram that depicts a query processor running on top of a backend DBMS.

---

[7]There are several "camps" of symbology; we are using the Yourdon notation for this discussion.

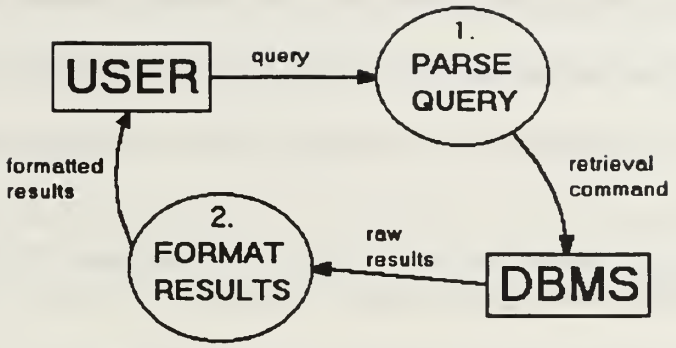


Figure 5. Example Dataflow Diagram

In this example, data (the user's query) flows from the USER to Process 1. The parsed query is then passed to the DBMS, which is external to the query processor in this example. The external DBMS then returns a result, which is formatted by Process 2. and then passed back to the user. The sequence in which the functions of the system are carried out is specified in the model only through the availability of data for each given process. All processes that have data available may theoretically execute simultaneously. For example, in Figure 5 if the user has entered another query while the first one is still being executed, both Process 1. and Process 2. could be running simultaneously.

2. Visual Dataflow Programming

Shu defines a *visual programming language* as "a language which uses some visual representations (in addition to or in place of words and numbers) to accomplish what would otherwise have to be written in a traditional one-dimensional programming language." (Shu, 1988, p. 138) Dataflow diagrams are inherently visually oriented. A graphical dataflow programming language allows construction of dataflow diagrams that

are not simply models but are directly translatable (by the computer) into executable code. Davis and Keller discuss the advantages of using a graphical representation for dataflow programs and give the following four main reasons (Davis, 1982, pp. 26-27):

- A dataflow graph conveys the mental image which suggests conceptually the data dependencies and flows between nodes.
- Dataflow programs are easily composable into larger programs.
- Dataflow programs avoid describing a specific execution order; they describe dependencies instead.
- Graphs can be used to attribute a formal meaning to the given dataflow program.

The two dimensional representation and the use of the value oriented computation method also help to increase the understandability of graphical dataflow programs (Washington University, 1986, p. 1).

Figure 6 is an example of a program fragment written in the graphical dataflow language Prograph. This fragment represents the equation $y = mx + b$. The values for

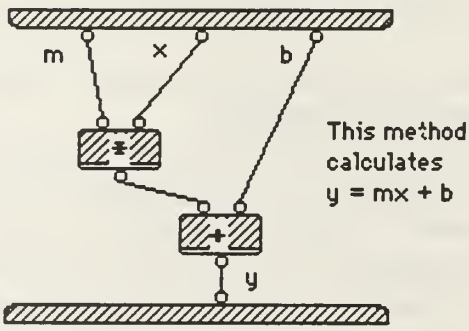


Figure 6. Dataflow Program Fragment

m, x, and b are expected as inputs, and the value for y is generated as the output. Prograph is a fully functional, object-oriented programming language based entirely on the graphical dataflow representation. Prograph is one of the few graphical dataflow languages that have been developed, and to the best of our knowledge, is the only commercial dataflow programming language on the market today. Our DFQL interpreter is written in the Prograph language. Prograph and how it was used to implement DFQL is discussed in detail in the implementation section of Chapter III.

III. DESCRIPTION OF DFQL

A. CONCEPT

DFQL is a visual relational algebra to be used for the manipulation of relational databases. It has been designed with sufficient expressive power and functionality to allow the user to easily express database queries. As such, DFQL is relationally complete and includes an implementation of aggregate functions. A facility is provided for the user to easily create his own DFQL operators, thus allowing great extensibility. Orthogonality has been stressed in the design of the language. The concentration on orthogonality provides a clarity of definition and lack of ambiguity that is missing from most other query languages (both visual and textual). Overall, the intent has been to provide the user with a simple to use, yet powerful and extensible tool to implement database queries at all levels of complexity.

DFQL has been developed as a token model graphical dataflow language. The use of the token model (Davis, 1982, pp. 27-31) implies that each of the defined operators are designed to operate on a stream of tokens over their lifetime. Our language does not allow the specification of iteration or recursion; each operator will execute once over the life of the given query. Iteration and recursion could be added to our language well within the dataflow paradigm. However, we feel iterative and recursive dataflow structures are not necessary for querying the database.

Queries are defined by the user connecting the desired DFQL operators graphically on the computer screen. The arguments for the operators flow from the bottom or "output node" of the operator to the top or "input node" of the next operator. Operator execution is controlled simply by the presence of the requisite input data for that operator's execution. When the data becomes available the particular operator may execute or *fire*. If there were facilities available, all fireable operators could be executed simultaneously. In our present implementation, only one operator is executed at a time since the system is being run on a single processor. The structure of DFQL queries directly mimics that of standard dataflow diagrams. The specifics of how this structure is implemented for DFQL are discussed in the following sections.

1. DFQL Operators

All DFQL operators have the same basic appearance. This has been done in order to enhance the orthogonality of the language. Each operator is made up of three types of components: *the input nodes, the body, and the output node*. A sample operator (with no name) is shown in Figure 7 below. The input nodes are where the data required

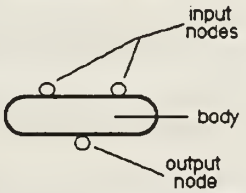


Figure 7. Operator Construction

by the operator enters. They are represented by small circles that are then connected to other operators by lines drawn by the user. The body of the operator is the large oblong to which the input nodes and output node are connected. For identification, the name of

the operator is displayed centered on the body. The output node is where the result of the operator exits. The output node may then be connected to other operators' terminals to pass the intermediate result along in the query.

The functional paradigm is fully supported by the DFQL notation. The inputs to each operator, or function, arrive at the input nodes of the operator and the result leaves from the output node. All of the operators of DFQL implement operational closure. This means that the inputs to the operators are relations and associated textual instructions, and the output from each operator is always a relation. Maintaining this concept is very important in the ability to understand large and complex queries. A lack of operational closure on query operators leads to complications in the formulation of complex queries. The complications are caused by the inability to orthogonally combine query operators when some operators yield relations as outputs whereas others yield some different type of data. Thus, the operators can only be combined in certain ways; if a scalar is output, it cannot then serve as input to an operator requiring a relation for input. It would necessarily be up to the user to construct his query in a manner consistent with all of the different data types output by the operators when operational closure is not enforced. This burden is especially great when the query being formulated is complex in its own right. Because all DFQL operators maintain operational closure, any output from a DFQL operator can be used an input to other operators.

There are two broad categories of DFQL operators that are based on their method of implementation. A *primitive* operator is one that has been defined directly in the native language of the DFQL interpreter. Primitives have a one-to-one

correspondence with an actual method in the implementation language of the interpreter. A *user-defined* operator is one that has been constructed by the user from primitives and possibly other previously created user-defined operators. The primitives can be further broken down into the categories of basic operators, other primitives, and display operators.

a. Basic Operators

In DFQL, the user is provided a set of basic query operators which he can then combine to build more complex operators as necessary. DFQL provides six *basic operators* derived from the requirements for relational completeness and also the requirement to provide a form of grouping or aggregation. Saying that a query language is relationally complete means that it has the expressive power of first-order predicate calculus. This is a common baseline measure of a query language's power of expression. For a query language to be relationally complete, the following five relational operations must be implemented: selection, projection, union, difference, and cartesian product. These operations are thus implemented as part of the basic set of DFQL operators.[8] Provision is also made for simple aggregation by including **groupcnt** (group count) as a basic operator. The **groupcnt** operator provides an easy solution to the universal quantification problem discussed in Chapter II. The basic operators and a corresponding translation into SQL are shown in Figure 8.

---

[8]Cartesian Product is not implemented explicitly; join is used for its implementation. This is in agreement with (Codd, p. 66, 1990).

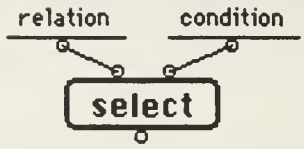
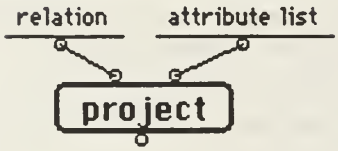
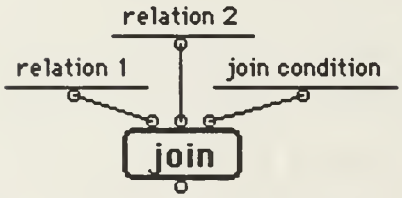
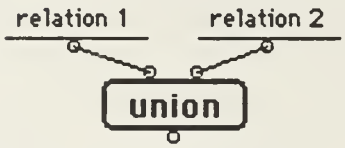
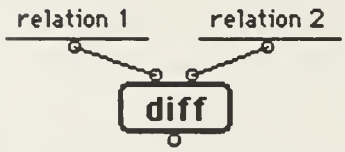
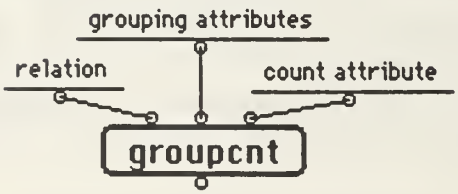
DFQL	SQL
	SELECT DISTINCT * FROM relation WHERE condition;
	SELECT DISTINCT attribute list FROM relation;
	SELECT DISTINCT * FROM relation1 r1, relation2 r2 WHERE join condition;
	SELECT DISTINCT * FROM relation1 UNION SELECT DISTINCT * FROM relation2;
	SELECT DISTINCT * FROM relation1 MINUS SELECT DISTINCT * FROM relation2;
	SELECT DISTINCT grouping attributes COUNT(*) count attribute FROM relation GROUP BY grouping attributes;

Figure 8. DFQL Basic Operators

A special notation is used to provide textual input to the DFQL operators. Text entered by the user shows up on the DFQL screen as an object with the text attached to an output node as shown in Figure 9. Text objects are the only DFQL syntactic item that

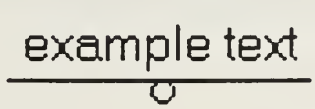


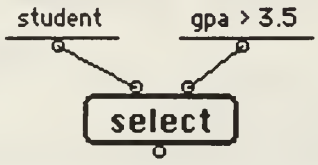
Figure 9. Text Object

generates an output other than a relation at its root. The text object can be interpreted in two different ways. If the text is the name of a relation, the output at the root can be thought of as an instance of that specific relation. If the text represents a condition, a list of attributes, or some other textual input to another DFQL operator, then the text is passed on to that operator as a textual argument.

(1) Select. This operator implements the relational algebra operation of database selection. The relational algebra notation for the select operation is as follows: $\sigma_{<condition>}(<relation>)$. The condition specifies which tuples should be retrieved from the given relation. The result of the selection operator (and all other DFQL operators) is a *proper relation.* By *proper relation* we mean a relation with no duplicate rows. Whenever we mention relations in this thesis we always mean proper relations, unless specifically stated otherwise. We make use of the explicit term when we wish to emphasize the characteristic of having no duplicate rows in a given relation.

An example of the use of the DFQL **select** operator is shown in Figure 10. This example retrieves all of the tuples in the STUDENT relation where the

GPA is greater than 3.5. As shown in the example, the condition input to **select** is an expression which must return a true or false value for each tuple in the source relation. The specification of this conditional statement uses the same syntax as in SQL. All of the tuples meeting the selection criteria form a new relation that flows from the output node when the operator is executed.

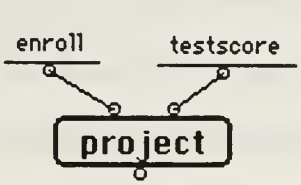


SNO	SNAME	ADDR	PHONE	GPA
S1	STU #1	ROOM 1	111-1111	3.85
S3	STU #3	ROOM 3	333-3333	3.75

Figure 10. Example DFQL Select

(2) Project. This operator implements the relational algebra operation of database projection. The relational algebra notation for the project operation is as follows: $\pi_{<\text{attribute list}>}(<\text{relation}>)$. The attribute list specifies the attributes that should be retrieved from the given relation. The syntax of the attribute list is simply the attribute names desired (non-case-sensitive) separated by commas. The result of the projection operator is required to be a proper relation made up of only those attributes specified in the attribute list. This requirement dictates the removal of what would otherwise be duplicate rows resulting from the removal of key attributes from the input relation. Figure 11 shows an example in which duplicate rows would result if they were not

eliminated by the operator. The Figure 11 example creates an output relation containing only the TESTSCORE attribute from the ENROLL table. In our example database, more than one tuple in the ENROLL relation contains the same TESTSCORE. As shown, these duplicate values are removed from the output relation by **project**, producing the required proper relation.



TESTSCORE
70
72
82
83
85
90
91
92
93
94
95
98

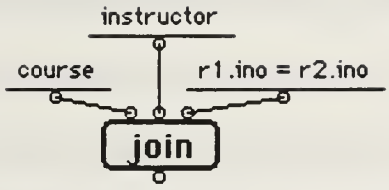
Figure 11. Example DFQL Project

The **project** operator can also be used to change the names of the attributes in the result relation. For example, in Figure 11 if we substituted "QUIZGRADE = TESTSCORE" for "TESTSCORE" in the attribute list, the result relation would have the same values, but the attribute would be named QUIZGRADE instead of TESTSCORE.

(3) Join. This operator is used to implement the relational algebra theta-join. We specify *theta-join* to stress that conditions other than equality of attributes may be used as arguments to the DFQL **join** operator. The relational algebra notation for the

join operation is as follows: <relation1>⋈$_{\text{<condition>}}$<relation2>. The result of the relational join is a relation consisting of all of the attributes from both <relation1> and <relation2>. The tuples of the result relation are the subset of the tuples of the cartesian product of <relation1> and <relation2> which satisfy the join condition. The join condition is specified using basically the same syntax as the WHERE clause in SQL. Range variables in the condition are limited specifically to **r1** (for <relation1>) and **r2** (for <relation2>). (These range variables need to be used only if the condition is specified on attributes with the same name in both of the input relations.) Normally the <condition> specifies some relationship between the attributes of <relation1> and <relation2>, but this is not necessary. Any <condition> that is a tautology will result in the cartesian product of <relation1> and <relation2> thus satisfying the requirement for cartesian product in the relationally complete set of DFQL operators.

Perhaps the most common use of the **join** is a special case commonly referred to as the *equi-join*. The equi-join specifies an equality condition between certain attributes of <relation1> and <relation2>. An example of an equi-join expressed in DFQL is given as Figure 12. In this example the COURSE and INSTRUCTOR relations are joined based on INO. The output relation produced contains all attributes from both the COURSE and INSTRUCTOR relations and conceptually is produced by selecting tuples from the cartesian product of COURSE and INSTRUCTOR where COURSE.INO = INSTRUCTOR.INO. The result of this **join** is a relation containing tuples for all of the courses taught combined with the instructor information for the instructor teaching that course.



CNO	TITLE	INO	INO1	INAME	PAY
CS05	COURSE # 5	I1	I1	INST #1	100000
CS10	COURSE #10	I2	I2	INST #2	50000
CS20	COURSE #20	I2	I2	INST #2	50000
CS15	COURSE #15	I3	I3	INST #3	47380.78
CS25	COURSE #25	I3	I3	INST #3	47380.78
CS30	COURSE #30	I3	I3	INST #3	47380.78

Figure 12. Example DFQL Join

The DFQL **join** retains all attributes of both of the input relations. Because all attributes are retained, special handling must occur when an attribute with the same name exists in both of the input relations. An alternative to retaining all attributes would be to discard one of the duplicated attributes as redundant. However, this approach places too much semantic meaning on the attribute name alone. For example, it is entirely possibly that we could have a NAME attribute in both the STUDENT relation and the COURSE relation (in our Appendix A example we use TITLE as the attribute for course name); a **join** to produce a relation of students and the courses they are taking should retain the NAME attribute from both the STUDENT relation and the COURSE relation. Although the two attributes have the same name, they represent two different things. For this reason, we have chosen to retain **all** attributes from both relations. Since the output relation may not have columns with identical attribute names, DFQL must provide a method of handling joins between relations that have attributes with the same

name. Our solution to this problem is to change the name of the attribute from <relation2> by appending a "1" to the attribute name. In the Figure 12 example, the attribute INO appears both in the COURSE and INSTRUCTOR relations. Thus, the result of the **join** has the two separate attributes INO and INO1. By taking this approach, no information will be lost, no matter what type of theta-join is performed.

A special case of the equi-join is the natural join. In a natural join one of the attributes that was used in the equality condition is automatically removed from the result relation. Natural join is not implemented as a primitive in DFQL since it does not provide any feature that cannot be produced from the provided primitives. However, if desired, natural join could easily be implemented as an additional primitive operator.

(4) Union. This operator implements database relational union. The relational algebra notation for the union operation is as follows: <relation1>∪<relation2>. The relational union is similar to but not as general as mathematical set union; the relational union requires that <relation1> and <relation2> be *union compatible*. Union compatibility means that when taken in sequence, the data types of the attributes in <relation1> and <relation2> must be compatible. This restriction is necessary because **union** does not create any more columns for the output relation. Both input relations must be of the same degree (have the same number of attributes) and the data types of corresponding attributes must be compatible in order to fit together in the result relation. Relational union produces a relation containing all of the tuples of both of the input relations (without duplication of rows).

The example shown in Figure 13 uses **union** to produce a relation containing the names of all the students and instructors from the example database of Appendix A. In this example we first **project** the names of the instructors and students and then take the **union** of the result. In the example database the attributes INAME and SNAME are of union compatible types. The renaming feature of **project** is also used in this example to change the result relation column name to "ALLNAMES." The default column name would have com from the first input relation and thus would have been INAME. The same query in SQL (without renaming) would be:

```
SELECT INAME
FROM INSTRUCTOR
UNION
SELECT SNAME
FROM STUDENT;
```

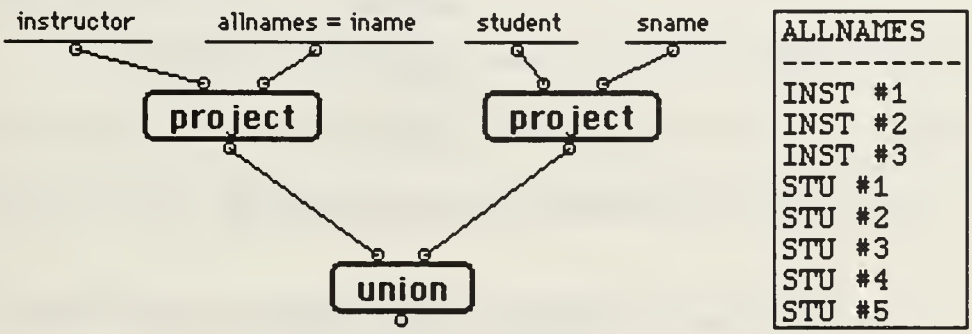


Figure 13. Example DFQL Union

(5) Diff. This operator implements database relational difference. The relational algebra notation for the difference operation is as follows: <relation1>-<relation2>. As with relational union, and in fact all set theoretic operators used in the relational model, **diff** requires that <relation1> and <relation2> be union compatible.

Relational difference returns as a result the relation that contains all the tuples that occur in <relation1> but not in <relation2>. Another way of looking at relational difference is that it "takes away" tuples from <relation1> that occur in <relation2>.

An example query using DFQL **diff** is given as Figure 14. This query returns as a result tuples representing courses that have no one enrolled in them. We first **project** CNO from both COURSE and ENROLL to produce two union compatible relations, and then we use **diff** to return the CNO's that were in the first relation (projected from COURSE) but not in the second (projected from ENROLL).

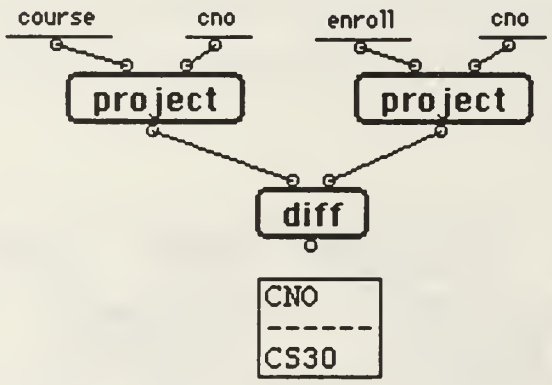
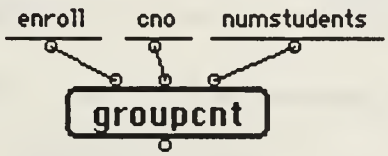


Figure 14. Example DFQL Diff

(6) Groupcnt. **Groupcnt** (short for group count) is defined as a basic operator in order to provide the user with some simple aggregation capabilities. Counting is especially important in allowing the user to easily formulate queries involving universal quantification. **Groupcnt** counts the number of tuples in a particular grouping specified by the user. For inputs, **groupcnt** requires a relation, a list of grouping attributes, and a name for the attribute in the result relation that will store the result of the count. The

grouping attributes may be a single attribute or multiple attributes separated by commas. The result relation will be made up of the attributes specified as grouping attributes along with the attribute name provided for the count attribute. The count attribute will be of an integer representing the number of tuples in the input relation that belong to each grouping specified by the grouping attributes. As a special case we allow the use of the keyword "ALL" as an argument for the grouping attribute list. If "ALL" is specified, **groupcount** simply counts all of the tuples in the input relation and as output produces a single attribute relation (using the attribute name specified for the count column) with a single tuple containing a count of the number of tuples in the entire input relation. This is consistent with the normal employment of **groupcnt**--since no grouping attributes were specified the entire relation is considered at once and there are no grouping attributes present in the output relation.

In Figure 15, **groupcnt** is used to produce a relation listing each course and how many students are enrolled in it. The result is produced by grouping the ENROLL relation by CNO and naming the counting attribute NUMSTUDENTS.



CNO	NUMSTUDENTS
CS05	4
CS10	3
CS15	3
CS20	2
CS25	2

Figure 15. Example DFQL Groupcnt

b. Other Primitives

We have provided several other primitives to perform special operations on relations. Most of these additional primitives perform operations that are at such a low level that the user would not be able to specify them as a user-defined operator. Several operators specified here as primitives could also be specified as user-defined operators. However, specifying an operator as a primitive allows us to take advantage of built-in functions of the underlying DBMS that we are running DFQL on top of. An example of this is the **intersect** primitive. Relational intersection can be defined in terms of **union** and **diff** ($R_1 \cap R_2 \equiv (R_1 \cup R_2) - ((R_1 - R_2) \cup (R_2 - R_1))$), however, many DBMS's provide a specific **intersect** operator. In order for DFQL to take advantage of this facility, we code **intersect** into the language as a primitive and use the underlying DBMS operation for its implementation. Also, specifying an operator as a primitive rather than as a user-defined operator slightly reduces the overhead required by DFQL to interpret the query. User-defined operators must be decomposed by DFQL into the primitive constituents prior to execution. This is avoided if the operator is simply coded as a primitive. However, due to the way that DFQL queries are executed, the difference in efficiency between primitives and user-defined operators is not great. Figure 16 shows the additional primitives.

(1) Eqjoin. The **eqjoin** operator is provided primarily to aid in the construction of user-defined operators. **Eqjoin** takes as arguments two relations (<relation1> and <relation2> and a list of attributes (<attribute1>, <attribute2>, etc.). The

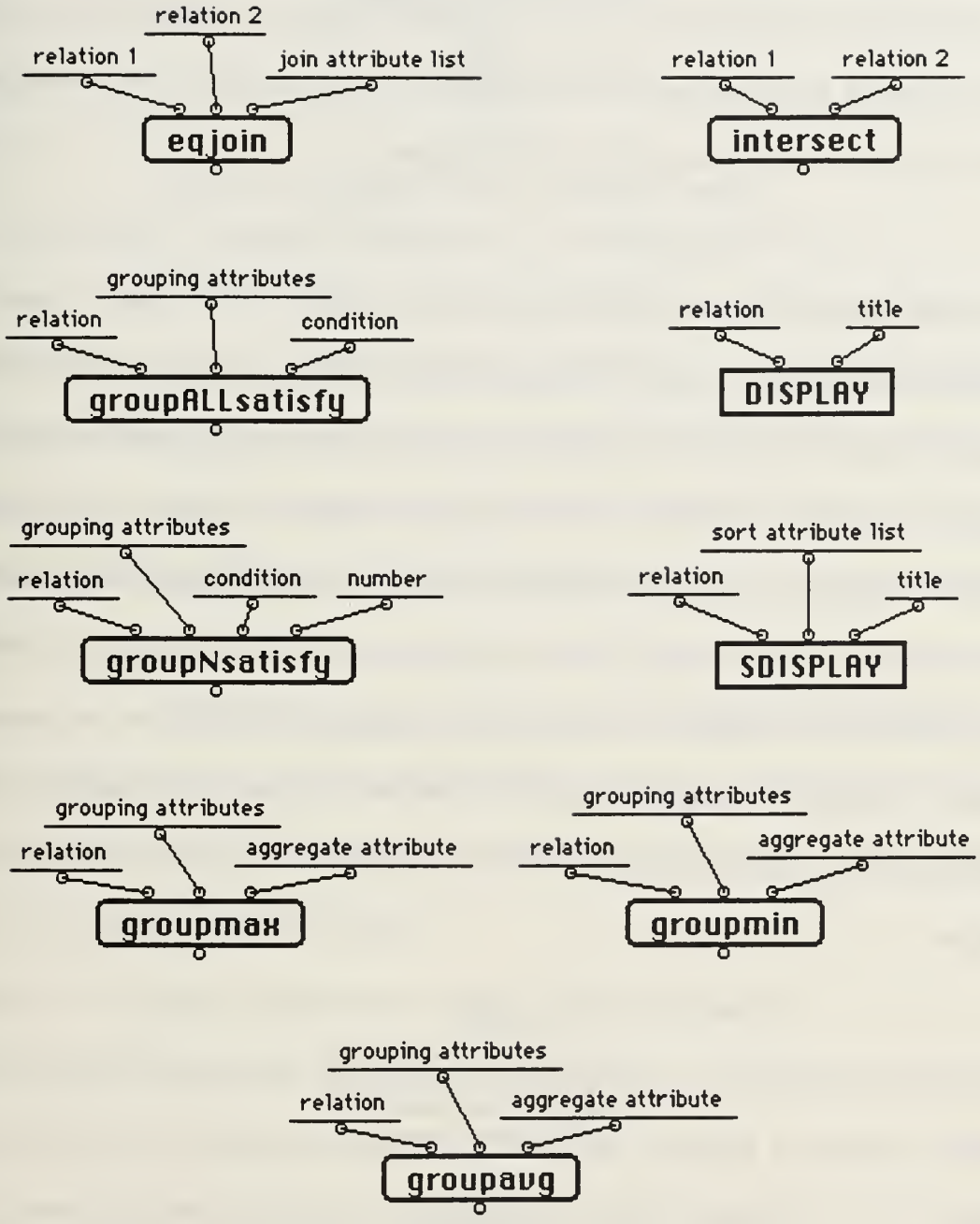


Figure 16. Other DFQL Primitives

attributes must occur in both <relation1> and <relation2>. An equi-join is then preformed by setting the join condition to r1.<attribute1>=r2.<attribute1> AND r1.<attribute2>=r2.<attribute2> AND etc. Thus the DFQL join condition is specified without explicitly including the equality statements. A later example (in the user-defined operator section) will show the utility of this operator.

(2) GroupALLsatisfy. This operator provides a simple way of introducing universal quantification into DFQL queries. The three inputs to **groupALLsatisfy** are the name of the input relation, a list of grouping attributes, and a condition statement that must be satisfied by all of the tuples in each group. The list of grouping attributes consists of the attribute names separated by commas. **GroupALLsatisfy** first groups the tuples according to the list of grouping attributes and then checks that all of the tuples in each group meet the condition specified. For each group that meets the condition, an output tuple is generated consisting of the grouping attributes. The result of **groupALLsatisfy** is a relation containing only those groups, as specified by their grouping attributes, where all the tuples in that group satisfy the given condition.

GroupALLsatisfy is used in the example of Figure 17 to retrieve the students who received 'A' grades in all of their classes. We specify this query on the ENROLL relation by grouping the tuples by SNO and specifying the condition as GRADE = 'A'. This means that all of the tuples in each SNO group must satisfy the condition that GRADE = 'A'. The result is a relation containing the SNO's of only those students with all 'A' grades.

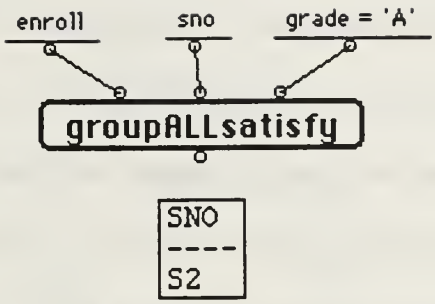


Figure 17. Example DFQL GroupALLsatisfy

(3) GroupNsatisfy. **GroupNsatisfy** is closely related to **groupALLsatisfy**. The only difference is that **groupNsatisfy** takes an extra input which allows the user to specify exactly how many of the tuples in the group need to satisfy the given condition in order for that group to be included in the result relation. This fourth argument to **groupNsatisfy** must consist of a relational operator (<, >, =, <=, >=, !=) and a number.

The example query in Figure 18 is much like the one in Figure 17, with the exception that we now use **groupNsatisfy** to retrieve those students who got more than two 'A' grades. This additional condition is specified by the ">2" entry as the fourth input argument to **groupNsatisfy**.

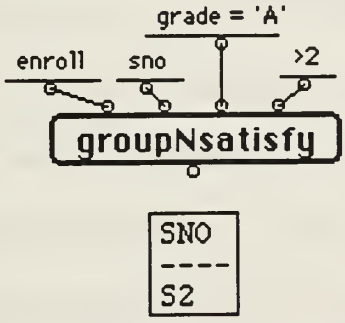
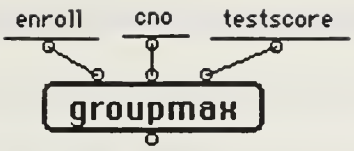


Figure 18. Example DFQL GroupNsatisfy

(4) Aggregate operators. Three aggregate operators are provided as primitives. Unlike **groupALLsatisfy** and **groupNsatisfy** which can be specified as user-defined operators, these aggregate operators cannot be formed from any combination of other operators--aggregate operators must be primitives. **Groupavg** is used to calculate the average of an attribute of a group of tuples in the input relation. Similarly, **groupmax** produces the maximum value and **groupmin** produces the minimum value of a specified attribute in each group. The three input arguments to all of the group aggregate operators are an input relation, a list of grouping attributes, and the attribute name to perform the aggregation on. The result relation consists of the grouping attributes and an additional column containing the result of the aggregate operation. This result attribute bears the same name as the aggregation attribute with the operation name prepended to it. For example, in Figure 19, we execute a DFQL query to return the maximum testscore for each course. The result relation is made up of a CNO attribute and a MAXTESTSCORE attribute. One tuple occurs for each course existing in the ENROLL relation. If the **groupmin** operator had been used the result relation would have a MINTESTSCORE column. Likewise, **groupavg** would produce AVGTESTSCORE as a result attribute.



CNO	MAXTESTSCORE
CS10	95
CS15	83
CS20	94
CS25	94
CS5	98

Figure 19. Example DFQL Groupmax

(5) Intersect. This operator implements database relational intersection. The relational algebra notation for the intersect operation is as follows: <relation1>∩<relation2>. The relational intersection requires that <relation1> and <relation2> be union compatible. The result of relational intersection is a relation containing only those tuples that occurred in both <relation1> and <relation2>. The implementation of **intersect** is identical to that of **union**. Two relations are taken as input arguments. The result relation is produced as discussed above. The data types of both of the input relations must be union compatible.

c. Display Operators

The display operators are not DFQL operators in the usual sense since they produce no output relation. The display operators are provided to allow the user to print the contents of relations on the computer screen. The most common use of the display operators is to print out the final result of a query. However, multiple display operators may be used in a single query to print out not only the *final* results but also

results at intermediate points in the query. This ability aids in debugging and formulating complex queries.

Due to the unique nature of the display operators they have a different shape than the rest of the DFQL operators. The display operators have square corners (and no output node) as opposed to the rounded corners of the rest of the DFQL operators. Their names are also displayed in all capital letters. These distinctions cause the display operators to be easily recognized in a query. The two display operators are **DISPLAY** and **SDISPLAY.**

(1) DISPLAY. The **DISPLAY** operator takes as inputs a relation and a text string to be used as a title. When **DISPLAY** is executed it causes the input relation to be printed out in tabular format. The text string that is input as the title is printed as the header for the output table. The title allows easy differentiation between printed results when more than one display operator was used in a query.

(2) SDISPLAY. **SDISPLAY** is used to produced a sorted printout of a relation. **SDISPLAY** takes as input a relation, an attribute list consisting of attribute names and, optionally, the order to sort them in, and a title.

The attribute list for **SDISPLAY** is different than the attribute lists for the other DFQL operators. Each attribute in the list may be followed by "ASC" or "DESC" to indicate whether the sort order for that attribute should be "ascending" or "descending." The order in which the attributes occur in the attribute list also is important. The "major" order columns are listed first with "minor" order columns

following. Thus, if we wanted to produce a listing of the ENROLL relation sorted first by CNO in descending order and then by GRADE in ascending order (within each course) the attribute list would be: "CNO DESC, GRADE ASC". This example is shown in Figure 20. The default ordering is ascending, so "ASC" actually never needs to be specified but may be included if desired for clarity. The title input operates the same way as in **DISPLAY**.




```
======================
SORTED DISPLAY EXAMPLE
======================

SNO   CNO    GRADE  TESTSCORE
----  ------ -----  ----------
S2    CS25   A              90
S4    CS25   A              94
S1    CS20   A              93
S5    CS20   A              94
S4    CS15   B              83
S5    CS15   B              82
S1    CS15   C              72
S1    CS10   A              92
S3    CS10   A              91
S2    CS10   A              95
S2    CS05   A              98
S4    CS05   A              93
S3    CS05   B              85
S5    CS05   C              70

14 records selected.
```

Figure 20. Example DFQL SDISPLAY

d. User-Defined Operators

One of DFQL's most important features is its extensibility through the use of ***user-defined operators***. With user-defined operators, the user can construct his own operators that look and behave exactly like the primitive operators provided in DFQL. The user can create operators for situations that are unique to his query needs. This flexibility is gained without a loss of orthogonality since user-defined operators are constructed by combining the provided primitives which have been coded to ensure maintenance of orthogonality.[9] The ability for the user to extend the query language with his own operations is an extremely powerful feature that is unique to DFQL.

A simple example of how a user-defined operator is constructed involves the **select** and **project** operators. In DFQL **select** and **project** are implemented as separate primitives. However, in use **select** and **project** often occur in pairs; first the selection is made and then a projection is done to retrieve only the specific attributes that are desired. An example of this would be to retrieve the SNO from the ENROLL relation where that student got at least one A. This query would be coded in DFQL as show in Figure 21.

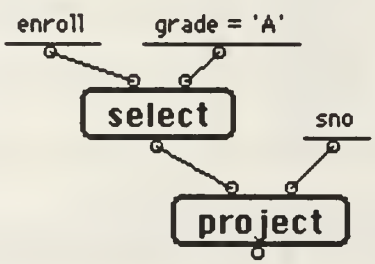


Figure 21. DFQL Select - Project Query

[9]User-defined operators may also contain other previously created user-defined operators.

Since combinations of **select** and **project** occur frequently it may be useful to have a single operator which combines these two operations. Figure 22 shows the specification of a new user-defined operator that does just this.

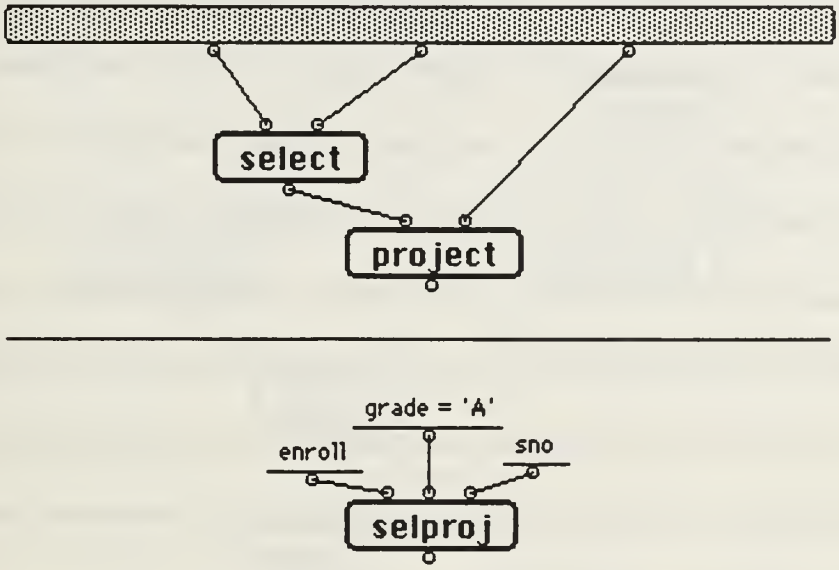


Figure 22. Creating a User-Defined Operator

The top part of the Figure 22 shows how the new operator is defined. The shaded gray rectangle at the top is called the "input bar." There are three "incoming nodes" on the input bar, hence the new operator will have three input nodes. The dataflow connections from the incoming nodes to the **select** and **project** operators are defined by the user. The result relation for the new operator flows out of the unconnected output node in the diagram. Once the specification of the internals of the operator is completed, the user must provide a name for the new operator. For this example, the new operator is called **selproj**. Once the user-defined operator is stored into DFQL, it may

be used just like any other operator. The bottom section of Figure 22 shows the same query as in Figure 21, but uses the newly defined operator.

Two advantages are gained from the utilization of user-defined operators. The most important advantage is that user-defined operators allow abstraction of complicated queries into manageable pieces that are easier to understand and use correctly. User-defined operators can thus greatly enhance the ability to write correct queries by relieving the user of the responsibility of repeatedly coming up with complex coding for commonly exercised queries. The complex code can be written once, tested, and converted into a user-defined operator that can simply be invoked without knowledge of its internal structure. Abstraction and encapsulation are modern techniques that are accepted universally in the field of software engineering but have never been put into practice in a query language until DFQL.

A second advantage of the user-defined operator is that it conserves space on the screen when the user is defining his queries. The lack of screen "real estate" rapidly becomes a severe problem for most graphically oriented applications. This problem is somewhat alleviated by user-defined operators.

As another example of a user-defined operator we include Figure 23, the definition of **groupALLsatisfy**, here coded as a user-defined operator (rather than as a primitive). This demonstrates the capability to define arbitrarily complex subqueries as user-defined operators. In fact user-defined operators may contain other previously defined user-defined operators to any level of recursion. This is possible because of the orthogonality enforced even when the user is allowed to create his own operators.

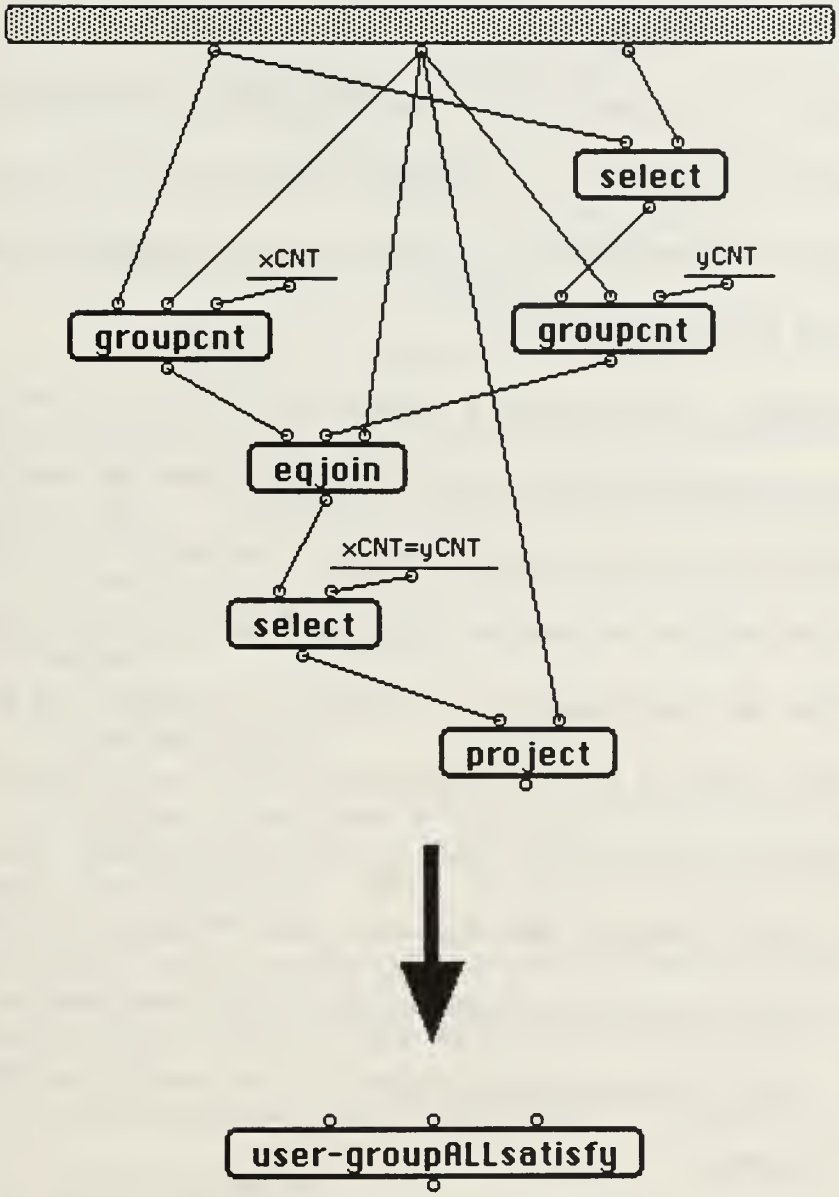


Figure 23. User-Defined Groupallsatisfy

Figure 23 is also an example of the amount of space that can be taken up by a subquery that is then condensed into a single operator.

2. DFQL Query Construction

Many of the general ideas behind DFQL query construction have been presented implicitly through the examples in the previous section. Here, we comment explicitly on some of the techniques used in DFQL query construction and on the benefits derived from the DFQL approach.

All DFQL queries exist as a dataflow program in which text objects and operators are connected by dataflow paths. The dataflow paths are represented as the lines in the DFQL query that connect the input and output nodes of the DFQL objects. Execution of the query can be visualized as flowing from the top of the diagram to the bottom.[10] When the input arguments to an operator are available, that operator may execute or "fire" producing its output which will then flow on to the other connected operators. Since text objects have no inputs, they may fire at any time. Execution of the query continues until all input has been exhausted. Since DFQL does not allow recursion or iteration within a query, each operator will fire exactly once during the life of the query. The results of the query are displayed for the user by the **DISPLAY** and **SDISPLAY** operators.

An example of a complete DFQL query is included as Figure 24. This query uses the **diff** operator to return the SNO of students who did not receive any 'A' grades.

---

[10]There is no restriction on how operators are placed on the screen. Top-down placement is recommended for readability.

In this query, the user-defined **selproj** operator from Figure 22 is used. There are several other ways that the same query could be posed in DFQL by using some of the other

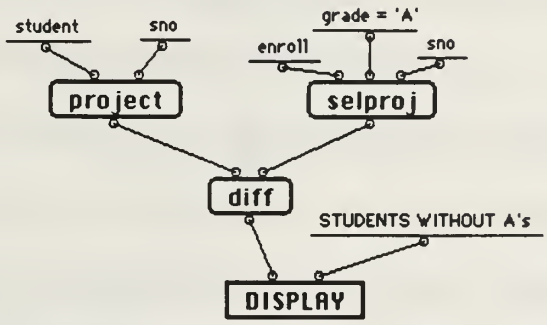


Figure 24. Complete DFQL Query

operators that we have discussed. One other method would be to use **groupNsatisfy** with the condition "GRADE='A'" and the count condition "=0". Depending on how this query is being used, as a part of a larger query or by itself, a user may prefer one method of expressing it to another.

a. Incremental Queries

The ability to easily build complex queries in an incremental manner greatly simplifies their formulation. DFQL provides two methods of support for incremental querying. The key to being able to construct queries incrementally is based on the operational closure property (Codd, 1990, p. 61). The output of any DFQL operator can be used as input to any other DFQL operator. This property can be used to great advantage in query construction.

To demonstrate the idea we will use a simple query for an example. The incremental query feature becomes of more value as the complexity of the query increases. In complex queries it becomes easier for the user to lose track of what he is

doing and what intermediate results that he has to work with. The example query is "List the names of instructors who taught CS10." To solve this query we can break it down into constituent parts as shown in Figure 25. First select all of the 'CS10' tuples from the COURSE relation. This result can be displayed to ensure we have what appears to be a correct partial answer. Next, join the partial result with the INSTRUCTOR relation to add the INSTRUCTOR information to the partial result. The new partial result can then be displayed. Finally, we can project INAME from the previous partial result to get the solution for our posed query.

Although the previous example is extremely simple, the value of the idea should be obvious. Perhaps an even more valuable advantage is gained through the use of *incremental query execution* as an aid in the debugging of a complex queries. When a large query is constructed, there are many possibilities for errors to creep in. Many of these errors are semantic and not syntactic; the DBMS will provide a result, but it will be erroneous. By going back through the query and looking at the intermediate results as it executes, the user is aided in finding where the flaw in logic occurred. In DFQL this practice is easily achieved. Given a complex query it is difficult to tell exactly where an error may have been introduced. DFQL allows the user to set a flag on any of the operators in a query to show the intermediate result at that point. For example, in Figure 26, the **join** operator is highlighted indicating that the user has selected this operator. Execution will stop at that point in the query and the intermediate result of the selected operator will be displayed. If that result was satisfactory, the user can search for the problem further along in the query. If that partial result was incorrect, the user can go

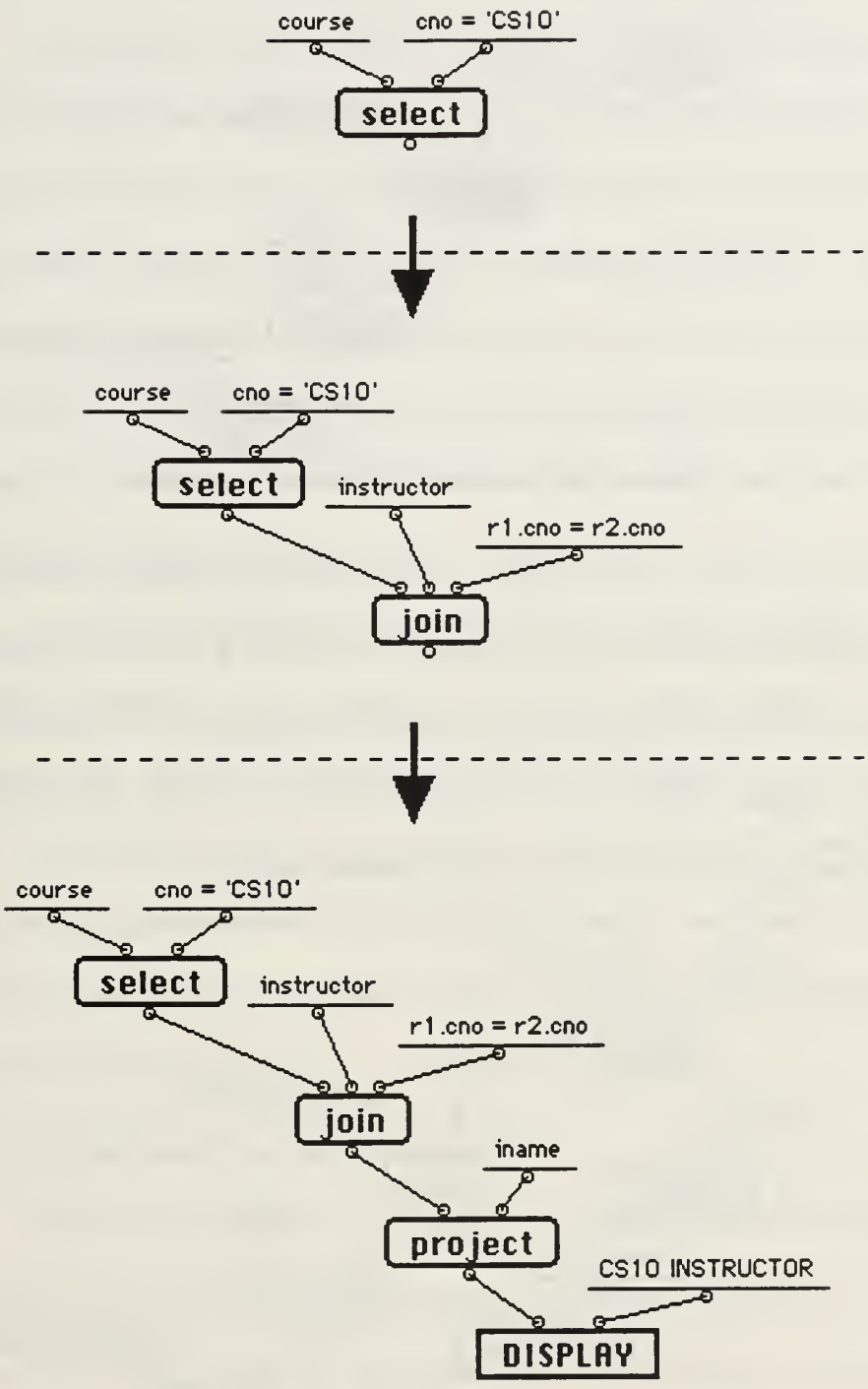


Figure 25. Incremental Query Construction

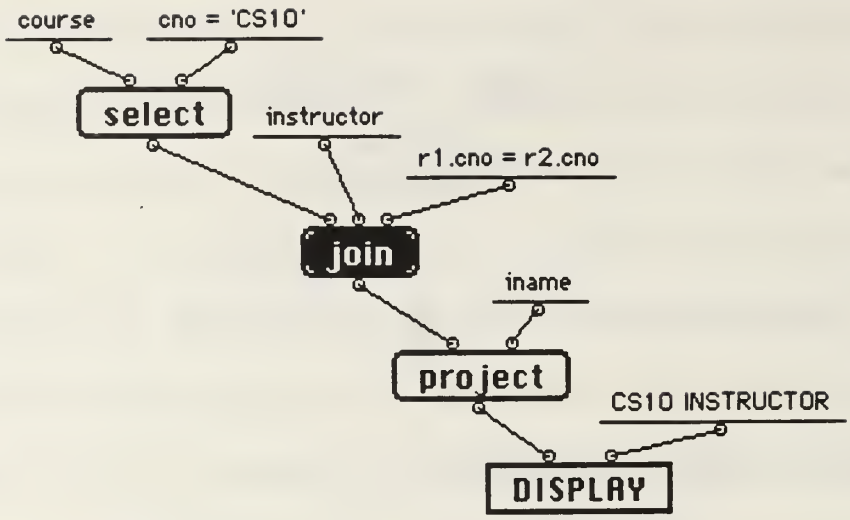


Figure 26. Incremental Query Execution

back and look at earlier partial results. Multiple display operators can also be used to report intermediate results at different locations in the query as shown in Figure 27. This method of analyzing intermediate query results has proven to be extremely useful in debugging complex DFQL queries. There is no easy way to even simulate this approach with complex queries in SQL due to its declarative nature.

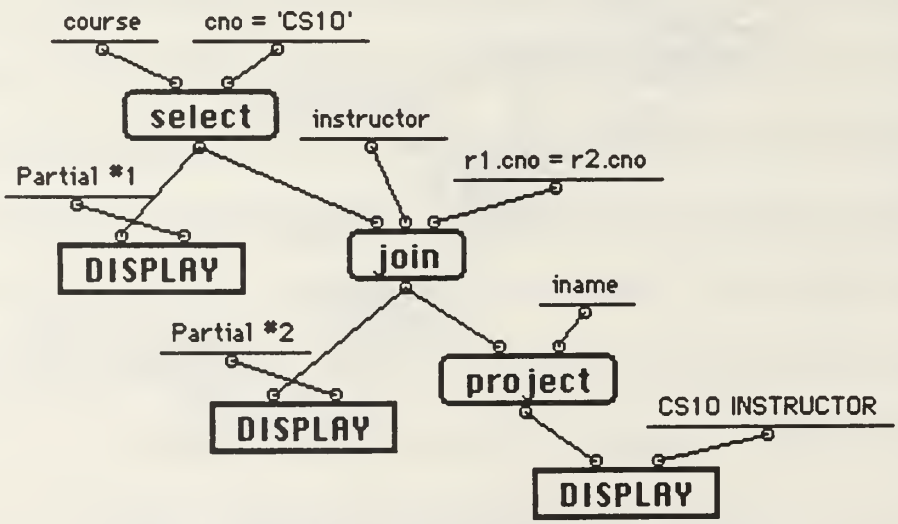


Figure 27. Use of Multiple Display Operators

b. Universal Quantification

The problem of expressing universal quantification in existing query languages has been discussed in Chapter II. DFQL provides a unique solution to this problem by starting with elementary counting operations that are easy to understand and then building on them to satisfy the requirements of universal quantification. The basic idea employed is that if all tuples in a relation or a group must satisfy some criteria, we first count the number of tuples that meet the criteria and then compare this number with the total number of tuples under consideration. If these two numbers are equal, then the universal quantifier has been satisfied.

The actual implementation of this idea is included in DFQL by the **groupALLsatisfy** primitive. A visual description of how **groupALLsatisfy** works is provided in Figure 23 where a user-defined operator was defined with the same functionality. The counting idea can be extended to supply other useful quantification type operators such as **groupNsatisfy**. The concept required to understand the idea of counting tuples is much simpler than that required to understand the logical idea of universal and existential quantification.

c. Nesting and Functional Notation

DFQL implicitly provides a nesting capability in the formulation of queries. Unlike SQL and block structured languages, however, there are no nesting constructs required in DFQL. Thus, DFQL requires no range variables or scoping rules; a good understanding of both range variables and scoping rules is necessary to code complex queries in SQL. The lack of nesting structures improves the readability and

orthogonality of the language. The idea of nesting, as implemented in SQL, is provided naturally in DFQL by having subqueries execute first and provide the arguments for later query operators. This is conceptually the same as executing nested queries in SQL from the "inside" to the "outside."

The use of functional notation for all of the DFQL operators greatly enhances orthogonality. The idea of relational operational closure discussed previously is naturally implemented through the functional paradigm. The use of operators that may take more than one input but produce only one output allows for their easy combination into user-defined operators as discussed in the previous section.

d. Graph Structure of DFQL Query

When a DFQL query is formulated, the visual representation of the query is a graph made up of operators (and text objects) as nodes and the dataflow paths as arcs. As such, the graph structure represents the relational algebra structure for the execution of the query. Having this structure provides two benefits: First, the internal operations of relational DBMS's are based on relational algebra. Thus, relational algebra can provide a common interface to a DBMS without the need of having a separate interpreter/compiler. Second, there is a large body of techniques that have been developed for the optimization of relational algebra expressions. Most SQL interpreters/compilers, for example, are not capable of performing optimization across levels of a nested query, but if the same query is expressed as a series of relational algebra operations it can then be optimized. (Dadashzadeh, 1990, p. 308)

By using a graphical, relational algebra approach to query formulation, we believe that the user is provided with a much more consistent and straightforward interface to the database. The advantages cited in the previous paragraph serve only to enhance the value of the graphical interface. Codd expressed a preference for relational calculus over relational algebra for a query language because of problems related to the DBMS's ability to optimize the queries (Codd, 1990, p. 62). The declarative approach of relational calculus has been preferred in the implementation of query languages in part in order to force the user to express his query in a single, large logical expression. For complex queries this large logical expression becomes difficult to correctly formulate. By using a graph structure of relational operators, the query can be more easily globally optimized than can be combinations of partial queries in a textual block structured language. In fact, the work of Dadashzadeh in converting SQL queries into relational algebra graphs for optimization purposes, results in structures quite similar to DFQL queries (Dadashzadeh, 1990).

B. USER INTERFACE FOR DFQL

DFQL and its graphical interface has been implemented on an Apple Macintosh. The general characteristics of the user interface follow the guidelines that Apple has established for Macintosh programs (Apple, 1985, chpt. 2). Basic operation of the program depends heavily on use of the mouse (or other pointing device) and pull-down menus. Every attempt has been made to make the user interface as friendly as possible.

Since ease-of-use is the most important goal of the DFQL language itself, ease-of-use of the interface is considered very important also.

In this section we provide an in-depth discussion on how the user interacts with the DFQL interpreter to formulate and execute his queries. We assume that the reader is familiar with such terms as "clicking", "double-clicking", and dragging with the mouse and "pressing a button" (on the screen).

1. Starting The Program

Upon startup, a title screen is displayed while program parameters are initialized. A dialog box is drawn to inform the user at the completion of the initialization phase. At this point the user is presented with the screen shown below as Figure 28. The **DB INTERFACE** window is the main window of the DFQL interpreter application. This window may be moved and resized anywhere on the screen that the user desires, but it may be closed only by quitting the DFQL application.

2. DB INTERFACE Window Items

a. Buttons

Several buttons are provided directly in the **DB INTERFACE** window for commonly required functions. Operator construction buttons are provided for the five required relational operators (**join**, **select**, **project**, **union**, **diff**), **groupcnt**, and **DISPLAY**. When one of these buttons is pressed, its related operator appears in the upper left corner of the drawing area in the window as shown in Figure 29. From this position the operator can be repositioned as desired by the user. (This procedure is discussed in the

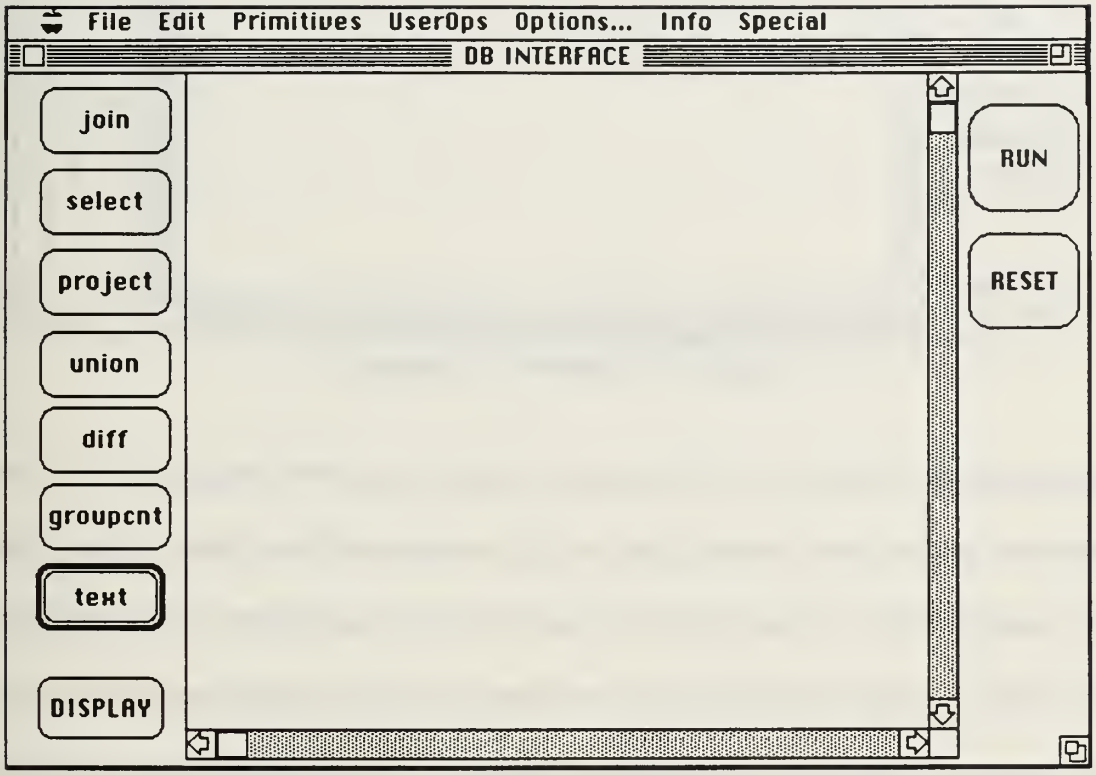


Figure 28. DFQL Main Window

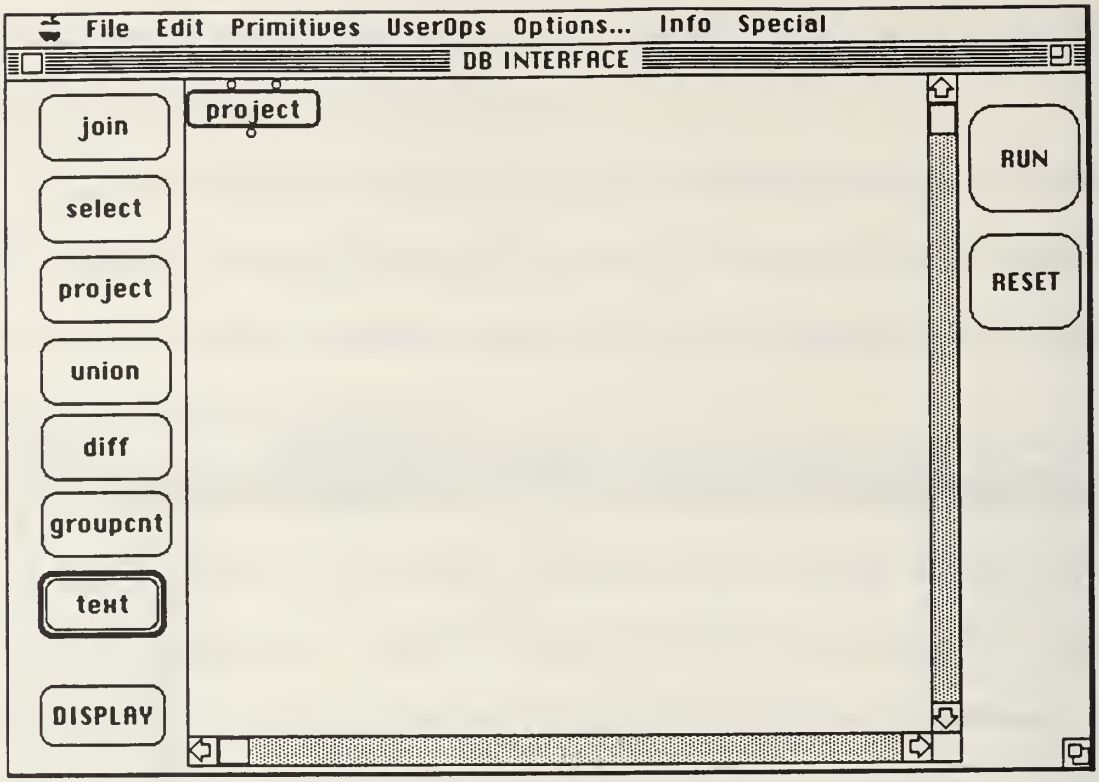


Figure 29. Operator Creation

Drawing Area section below.) There is no harm if the operator is not moved from this position and another one is created. Each of the operators will continue to exist. Even if one covers another, the operators can be peeled off of each other with no problem. Along with the operator construction buttons there is also a text object button. When this button is pressed a dialog box (as shown in Figure 30) is opened for the user to enter the character string for the text object. When the user clicks the OK button or presses the **return** key on the keyboard, the text object is created and appears in the same position on the screen as newly created operators. The length of text displayed can be limited in order to not clutter the screen. Truncation of the displayed text is indicated by trailing "..."--the change in the display format does not affect the actual value of the text string.

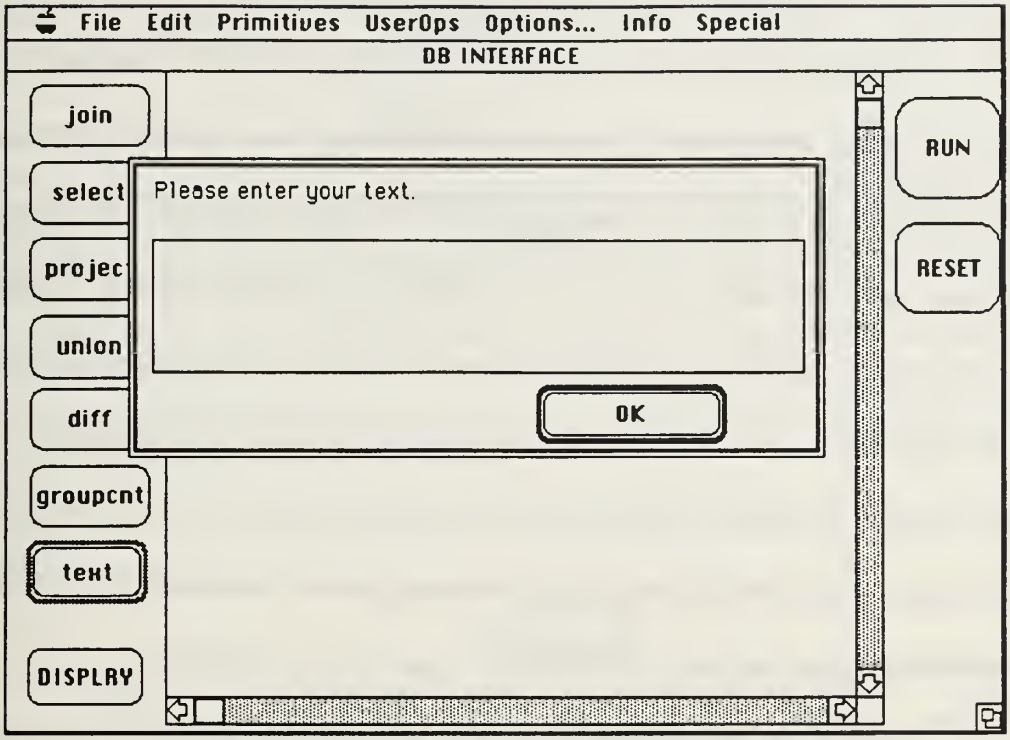


Figure 30. Text Object Creation

The **RUN** button executes the query that is currently displayed in the drawing area. **RUN** will first check that the query graph displayed is constructed correctly; it ensures that all input nodes are connected, for example. Then the query will be sent off to the backend DBMS for processing. Results returned from the database will be displayed in a separate result window. The **RESET** button clears the current query from the drawing area and from the computer's memory. **RESET** can be used to set up another query when the user has no desire to save the query that is currently on the screen.

b. Drawing Area

The *drawing area* is the portion of the window that is bounded by the horizontal and vertical scroll bars. This area starts out blank and is used to graphically construct the DFQL query from the various operators and text objects. As the query becomes larger, the scroll bars may be used to position it in the drawing area so that the portion of interest is displayed. In order to move an operator or text object within the drawing area the user clicks on and then drags the object to the desired position. While dragging the object, an outline is displayed that shows the position of the object as it is being moved around the drawing area. When the mouse button is released the object (and any connected dataflows) are redrawn in the new position.

Along with the dragging of objects there are several other operations that can be performed on the DFQL query in the drawing area. Double-clicking on an operator will bring up a help window describing that operator. The help information for the DFQL primitives is coded into the system. Help information for user-defined operators is entered by the creator when the operator is defined. Help information appears in a dialog box as shown in Figure 31. Double-clicking on a text object opens up an editor for that object's text string. This editor supports all of the Macintosh's normal text editing functions such as cutting and pasting text from the Macintosh clipboard. When the editing dialog box is closed, the text for the object is replaced with the new string.

In order to construct a DFQL query, the query objects must be connected with the desired data flows. These flows are represented in the interface as straight lines

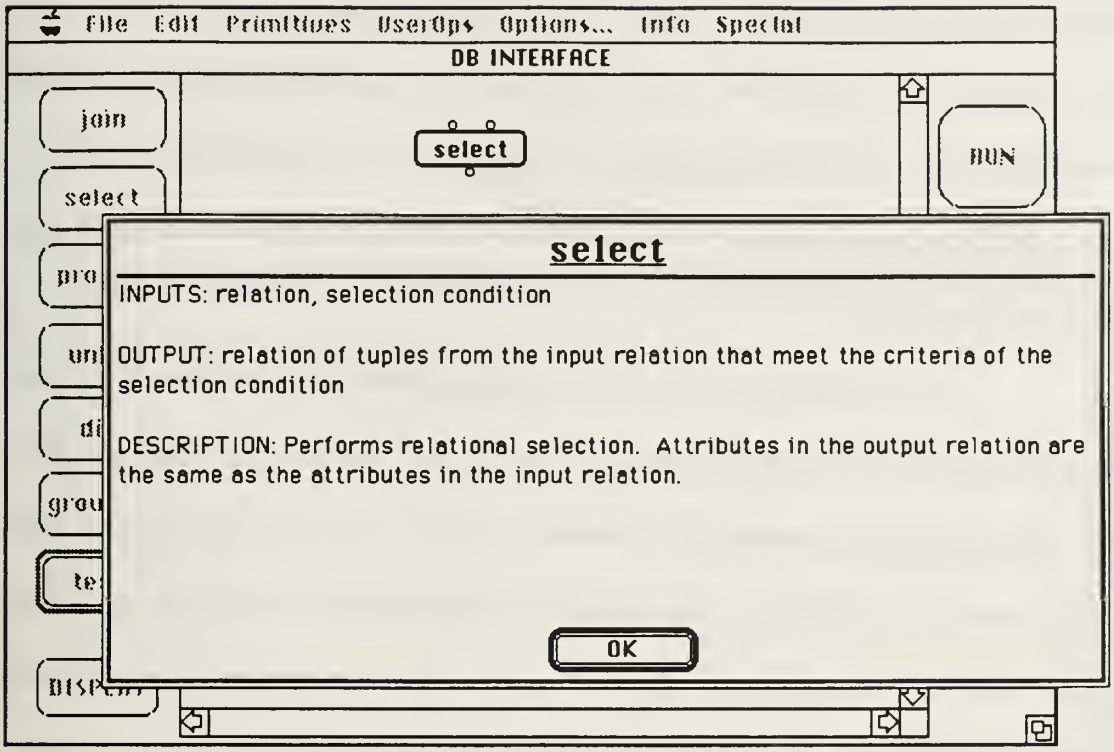


Figure 31. Example Select Operator Help

that connect the output node of any given object to the input node of another object (or objects). To draw these lines, the user must click the mouse pointer on either an input or output node. Once the mouse button has been released, a rubber-band line will be drawn from that node to the current position of the mouse. Clicking on the input or output node of another object will connect the dataflow line from the originating node to the newly indicated node. DFQL does some checking to ensure that connections make sense. For example, attempted connections from input to input or output to output are detected and an error message is produced stating that the attempted connection "did not make sense." This level of error checking is somewhat rudimentary, however. DFQL will not flag *cycles* created in the query graph at construction time. An error message

will be produced when the query is executed. Clicking the mouse in an empty portion of the drawing area will turn off the rubber-band line if the user has decided not to make a connection after all. Since an input node may have *only one* input dataflow, if the user connects a dataflow line to an input node that already had one, the previous dataflow line is deleted automatically.

If the mouse is double-clicked on an output node, the columns of the relation passing out of that node are displayed. In this way, the user can determine what attributes may be used by operators subsequent to that point in the query graph. This assistance is very important in the construction of large queries in which the attributes become hard to keep track of. Also, when user-defined operators are used, it is important to be able to easily determine what the names of the attributes are that the operator produces.

3. Query Results Window

The Query Results window displays the result of the DFQL query. The results are displayed in the format returned by the backend database system. An example of a displayed query result is included as Figure 32. The contents of the results window may be edited with any of the Macintosh's normal editing functions (cut, copy, paste, and clear). The results may also be sent to the printer. Scroll bars are provided in the result window in order to display queries that generate results that are larger than the viewable area. The query results window may be moved, resized, and closed as the user desires. For example, in Figure 32, the results window has been moved so that the query in the DB INTERFACE window is visible. If DFQL is being run on a system with a large

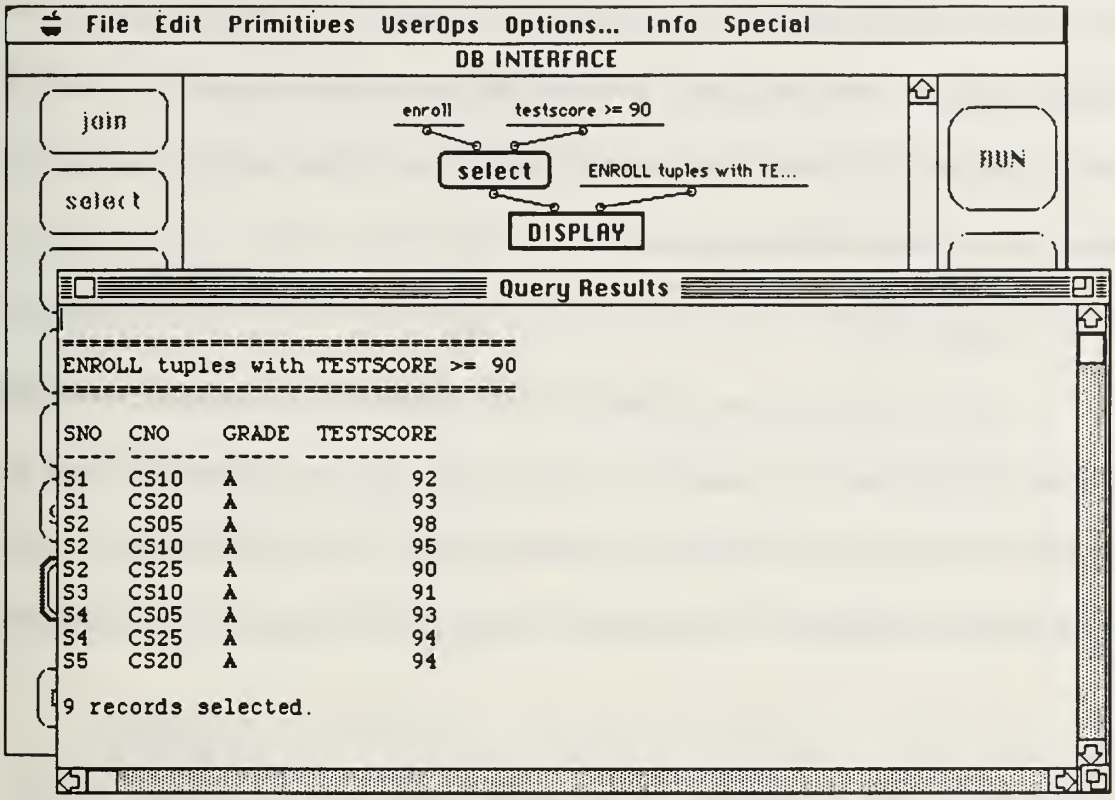


Figure 32. Query Results Window

monitor, the Query Results window could be moved and left open while queries are formulated in the DB INTERFACE window. If there is room, the results window can be resized in order to display more of the result at once. The Query Results window is activated when the query is run from the DB INTERFACE window. If the Query Results window is closed, it will not be reopened until the next query is run.

The Query Result window is also where error messages about the current query will be returned to the user. All errors relating to the DFQL query, with the exception of the graphical construction type errors mentioned previously, are trapped by the backend DBMS. These errors are then passed back to the user through the Query Results window.

Since the error messages may reference temporary views created by the DFQL interpreter, an option is provided for the display of the actual SQL query that was sent to the backend DBMS. This feature allows for easier debugging of the DFQL query; its necessity is discussed in the Implementation section.

4. Menu Items

The menu bar displayed at the top of the screen is an omnipresent feature of all Macintosh programs. Its presence and design is one of the requirements dictated for Macintosh user interfaces by Apple (Apple, 1985, p. I-51). The DFQL interpreter menu bar is displayed as Figure 33. In a Macintosh environment the menu bar is a separate



Figure 33. DFQL Menu Bar

entity from the window currently being displayed. For that reason, the items listed in the menu bar usually remain the same throughout execution of the given application no matter what window is currently being displayed; any items that are not applicable at a given time are made not selectable. Any item in the Macintosh user interface that selectable is indicated to the user by being displayed at reduced intensity, commonly known as being "grayed out." The DFQL user interface menu items are discussed individually below.

a. Apple

This menu is a standard Macintosh menu that has no real relation to DFQL as an application. It provides access to Macintosh utilities called "Desk Accessories" and should be accessible at all times (Apple, 1985, p. I-54). The only DFQL specific item in the Apple menu is the "About..." item. When this item is selected a title and information window for DFQL is displayed.

b. File

The file menu (Figure 34) also has a standard Macintosh design (Apple, 1985, p. I-55), but is application specific in its functionality. Our file menu has six items

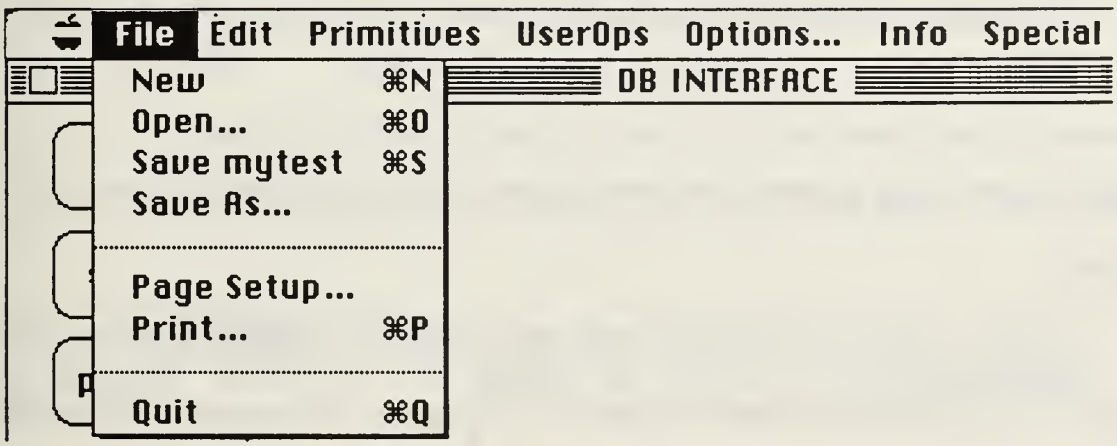


Figure 34. File Menu

which follow the Macintosh user interface guidelines. The **New** item resets the system for the user to enter an entirely new query. The **Open...** item allows the user to retrieve a previously saved query from disk. When **Open...** is selected, a dialog box is presented from which the user can select the stored query for retrieval (Figure 35). Only query files are displayed for selection. Once a query file has been selected, it is immediately loaded,

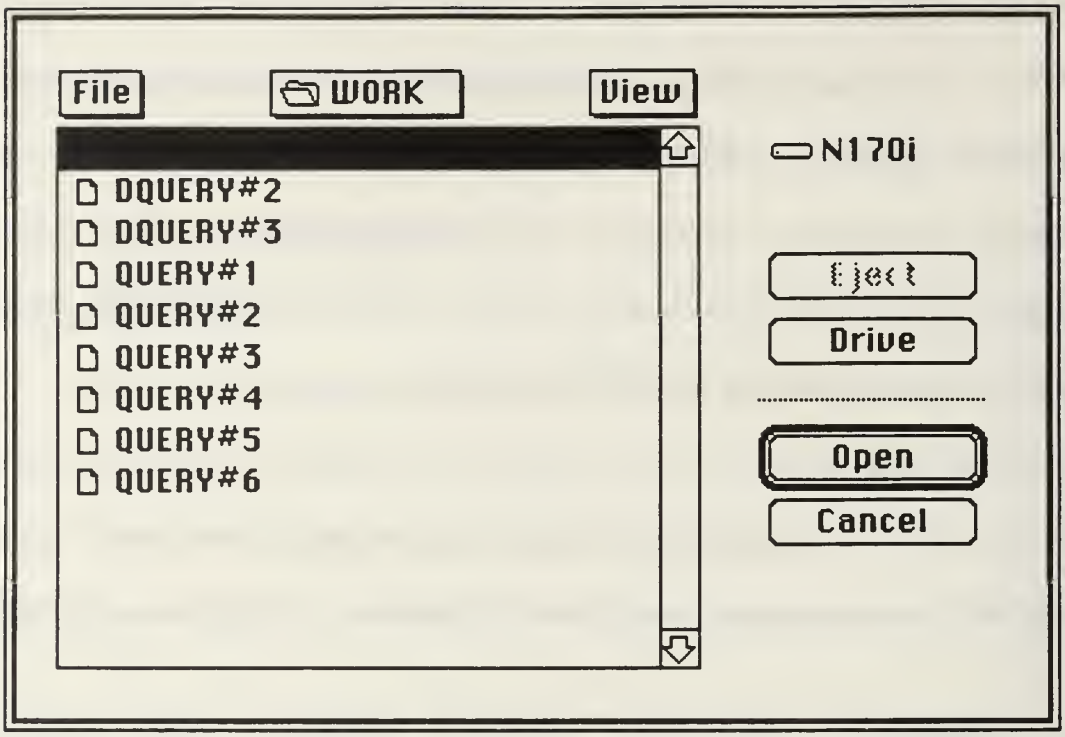


Figure 35. Open... Dialog Box

and the stored query appears in the drawing area in the window ready for execution or editing.

The **Save** option (shown in Figure 34 as **Save untitled**) stores the current query onto disk with the name that is currently displayed. For example, **Save untitled** would create a query file actually named "untitled". When a query is retrieved using the **Open...** command, its name is retrieved also and will appear in the **Save** menu item. If the current query was a "new query" and thus had no name **Save untitled** will be displayed as the **Save** option. The user can use the **Save As...** menu item to name the file. This option displays a file naming dialog box. If the user enters a name that is already in use he is asked whether or not he actually wants to replace the previously

stored query. If not, a new name can be chosen. When an appropriate name has been chosen the query will be saved to disk.

The **Page Setup...** option is a standard Macintosh **File** menu item. **Page Setup...** runs a Macintosh routine which allows the user to change printer parameters such as the size of paper, print quality, and orientation. **Print...** is used to print out information from the front window of the application. For example, if the DB INTERFACE window is foremost then the DFQL query currently displayed in the drawing area will be printed. If the Query Result window is foremost then the text of the query result will be printed. The **Quit** menu item terminates the DFQL interpreter execution.

c. Edit

The Edit menu (Figure 36) is another of the Macintosh standard menus. It provides the text editing functions of **Cut**, **Copy**, **Paste**, and **Clear**. These edit

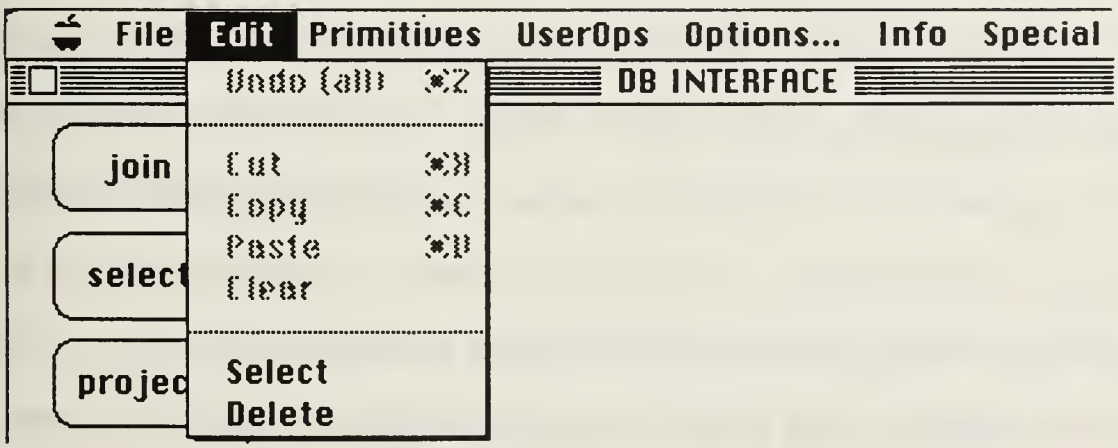


Figure 36. Edit Menu

functions are available whenever the user is editing a text item, such as when the Query Results window is displayed. An **Undo (all)** menu item is also provided. **Undo (all)** reverses the deletion of objects in the DFQL drawing area. It is only active immediately following the deletion of the objects. When **Undo (all)** is not available it is "grayed out".

The two remaining choices in the Edit menu are used to edit the DFQL drawing area. **Select** is a "checkable" item. By this we mean that it has two conditions--on and off. When **Select** is turned on, a check mark appears to the left of the **Select** menu item. While **Select** is on, clicking the mouse on a DFQL object in the drawing area will cause it to be "selected". This selection will be indicated on the screen by the object's color being inverted as shown previously in Figure 26. Selection of objects in the drawing area is a "toggle" type process. If the mouse is clicked on a previously selected object, the selection will be toggled off and the operator will return to its normal appearance. Selecting a DFQL object has two effects. First, it enables the **Delete** operation which is also an item in the Edit menu. **Delete** will delete all selected objects from the DFQL drawing area. Secondly, selecting a DFQL operator allows the user to retrieve intermediate results from the query. When an operator is selected and the **RUN** button is pressed in the DB INTERFACE window, the query will be executed up to and including the selected operator. The result of this operator will then be displayed in the Query Results window. When the **Select** menu item is turned off (by choosing it while it is check marked) all of the currently selected DFQL objects are returned to their non-selected state.

d. Primitives

The Primitives menu (Figure 37) allows the user to select primitives that are not provided by a button in the DB INTERFACE window. These primitives include:

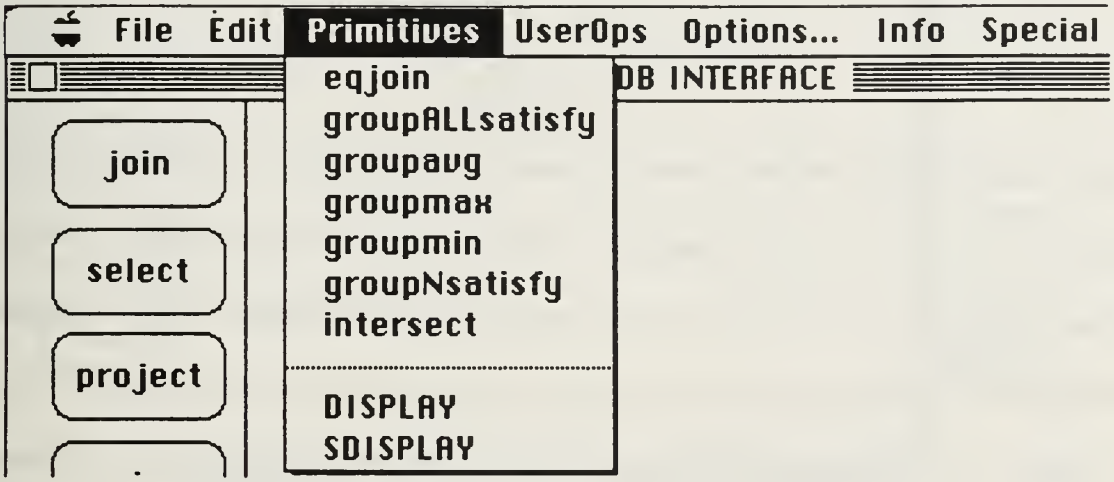


Figure 37. Primitives Menu

eqjoin, **groupALLsatisfy**, **groupavg**, **groupmax**, **groupmin**, **groupNsatisfy**, **intersect**, **DISPLAY**, and **SDISPLAY**. When one of these menu items is selected the effect is the same as pushing one of the primitive buttons on the DB INTERFACE window. The desired operator appears in the upper, left corner of the drawing area and is ready to be incorporated into a DFQL query.

e. UserOps

The UserOps menu (Figure 38) is provided to enable the user to define and manipulate user-defined DFQL operators. The **New** menu item places the DB INTERFACE window into user operator definition mode. This mode disables the window's normal menu and button items and adds several operator definition items to the



Figure 38. UserOps Menu

screen. This operator definition mode is shown in Figure 39. The desired internal structure for the new user-defined operator must exist in the drawing area before choosing the New menu item. Connections in the drawing area may be changed while in operator definition mode, but no operators may be added or deleted since all of the required menu and button items are disabled. The most obvious added item in this operator definition mode is the "input bar" at the top of the screen. This bar is used to define where the input data to the user-defined operator will be sent internally. Clicking the mouse on the input bar will create additional input nodes for the user-defined operator. If too many nodes are created by mistake, input nodes can be removed by checking the Delete Input check box on the right side of the window. Whenever this box is checked, clicking on the input bar will delete the input nodes (and all internal connections to them) from the drawing area. Once the desired number of input nodes are created, they must then be connected to the desired operators in the drawing area. These dataflow connections are made in the same manner as on the normal DFQL editing screen. All input nodes of the operators inside the user-defined operator must be connected. Also, there may be only one unconnected output node in the user-defined operator. This single node becomes the

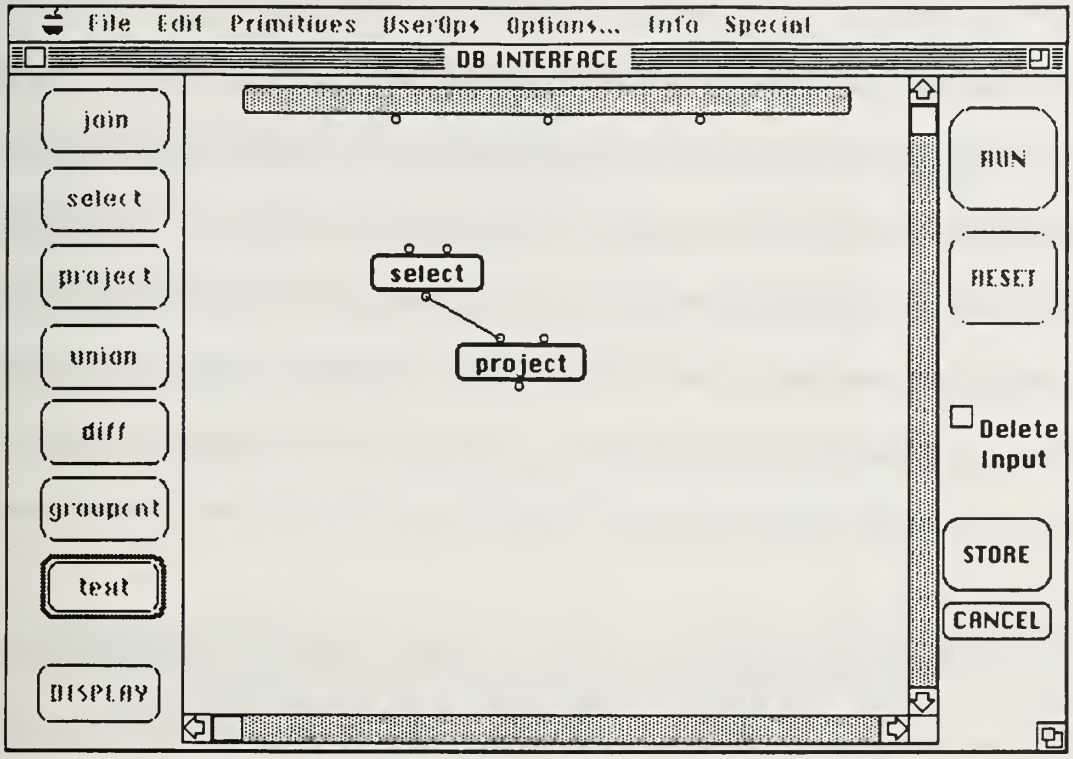


Figure 39. User Operator Definition Window

output node for the entire user-defined operator. Figure 39 shows **select** and **project** in the drawing area ready to be connected to three nodes that have been created on the input bar.

There are two active buttons provided in user-operator definition mode: **Store** and **Cancel**. **Cancel** restores the screen to the normal DB INTERFACE window by eliminating all of the user-operator definition items and reactivating the normal DB INTERFACE buttons and menus. The items that were in the drawing area in the operator definition mode will still be present with the exception of the input bar. **Store** first checks the user-defined operator query graph to ensure that all necessary connections have been made and then asks the user for a name for the new operator and a description that

will be used as help for the operator. The operator's name is checked for uniqueness among all previously defined user operators; the new name must be unique. When entering the help for the operator, the user should list what type of argument is expected for each input node, what relation is produced from the output node, and also provide a brief description of what the operator does. In order to make the help information more easily readable, the user may insert **carriage return** characters by using the **Option-Return** key combination on the Macintosh. Once all of the requisite user input is received, the new user-defined operator is added to the list of currently stored user-defined operators.

The **Delete** menu item allows the user to delete stored user-defined operators from the system. The user is presented with a scrolling list of user-defined operators, as shown in Figure 40. When the desired operator is selected, by either

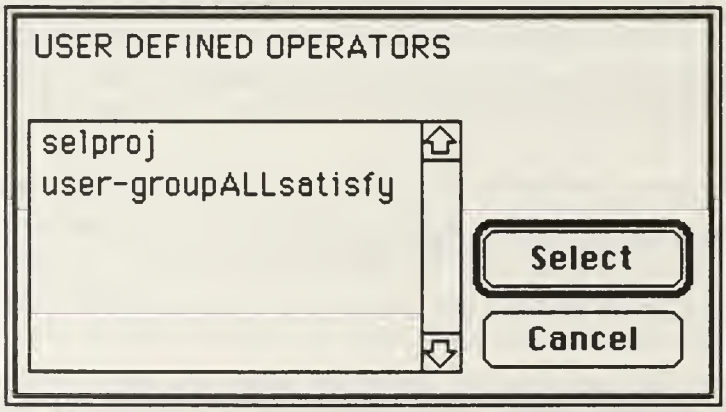


Figure 40. User-Defined Operator Selection

double-clicking on its entry or selecting it and then pressing OK, it will be deleted from the list of operators. The **Select** menu item presents the user with the same type of

scrolling list. **Select** is used to add a user-defined operator to a DFQL query. When the desired operator is chosen from the selection list, it appears in the upper left corner of the drawing area just the same as a DFQL primitive. There is no difference in the use and manipulation of the user-defined operator as compared to a primitive DFQL operator. The final menu option in the UserOps menu is **View**. **View** allows the internal structure of a stored user-defined operator to be displayed. An example of this display is Figure 41. The desired operator is selected through the use of a selection dialog as shown

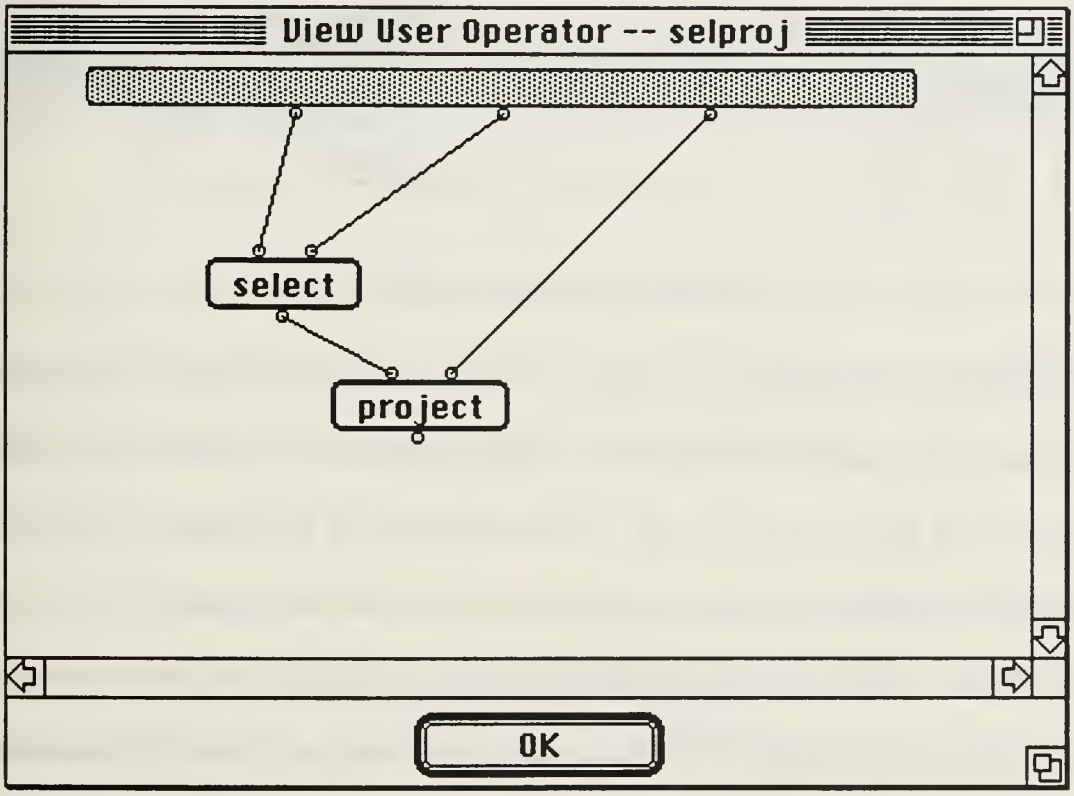


Figure 41. View User Operator Window

previously in Figure 40. The **View** feature is provided so that a user may "look inside" the operator to see how it was constructed. This is especially useful if the user-defined operator was provided by someone else. The user is *not* permitted to modify the operator, only look at it. In this way the integrity of the operator is preserved while still allowing some access to the internals for the user's purposes.

f. Options...

The Options... menu (Figure 42) provides the user with control over the operation of the DFQL interpreter. All of the choices provided in the Options... Menu

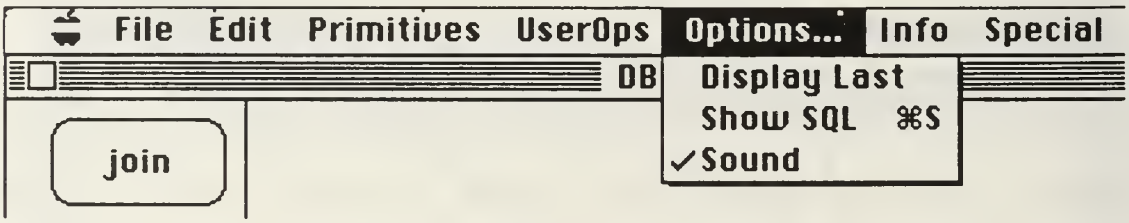


Figure 42. Options... Menu

are toggle items. When the item is active, or "turned on", a check mark is present next to the item. For example, in Figure 42 the **Sound** option is "on" whereas the **Display Last** and **Show SQL** options are "off". When **Display Last** is turned on, the output of the last DFQL operator executed will be displayed in the Results Window when the query is run. This is useful when incrementally constructing queries because it causes the display of the results without having to use a display operator. **Show SQL** causes the intermediate SQL code that is generated from the DFQL query graph to be displayed in the Query Results window along with the results of the query. This display can be used to troubleshoot any execution errors that are not directly apparent from the DFQL query

graph. Also, this option allows the DFQL interpreter to be used as a translator in which a DFQL query is input and a SQL query is output which could then be run on any SQL database system. The **Sound** option is included primarily for esoteric reasons. When selected, **Sound** causes certain easily recognizable sounds to be played at different key points during processing of the query.

g. Info

The Info menu currently has only one option, **Tables**. This option allows the user to retrieve information about what attributes exist for tables in any given relation



Figure 43. Info Menu

in the database. When **Tables** is selected a selection dialog is displayed from which the user can pick which table he is interested in. This action will bring up a dialog box displaying the attributes of the selected table as shown in Figure 44.

h. Special...

The Special... menu also has only one menu item, **ORACLE*Shell**. **ORACLE*Shell** starts up a separate application to provide the user direct access to the backend DBMS (in this case ORACLE). When **ORACLE*Shell** is selected the user is presented with a new window and menu bar that are specific to the ORACLE*Shell



Figure 44. Table Information

application. From this window the user may select SQL*Plus to start up the ORACLE SQL interpreter, as shown in Figure 45.

Once the SQL*Plus interpreter is running, the user may manipulate the database directly using any SQL command desired. This facility allows the user to add, delete, and update tuples in the database relations. Since the current version of DFQL is strictly a *query* language, these database functions are not provided in DFQL. By allowing the user direct access to the backend DBMS while still running under the DFQL environment this deficiency is somewhat mitigated. In order to return to the DFQL

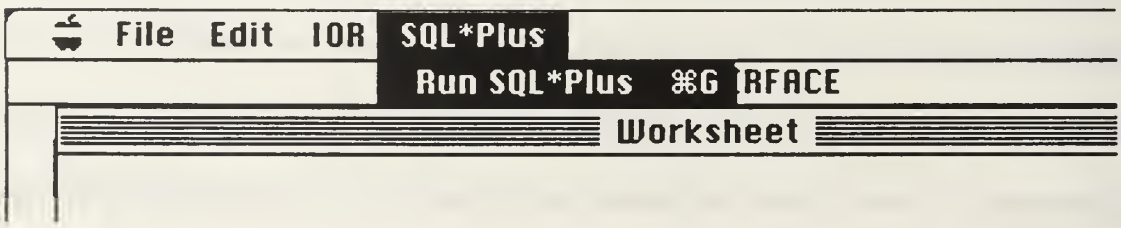


Figure 45. Starting the SQL*Plus Interpreter

interpreter, the user must first exit from SQL*Plus (by typing "EXIT" at the "SQL>" prompt) and then stop ORACLE*Shell by selecting **Quit** from the File menu. When the user exits from ORACLE*Shell, control is automatically returned to the DFQL interpreter.

C. IMPLEMENTATION OF DFQL

DFQL has been implemented on a Macintosh II/ci running version 6.0.7 of the Macintosh Operating System. The actual programming was done in the Prograph language (discussed further below). The backend DBMS used on the Macintosh is ORACLE for the Macintosh version 2.0. DFQL has also been operated on a remote ORACLE DBMS (version 6.0) running on a Digital Equipment Corp. Micro-VAX via a DECNET Ethernet connection. XLINK protocol is used to communicate database startup and shutdown commands to the ORACLE kernel running on the Macintosh. All features of DFQL which have been discussed in this thesis have actually been implemented.

1. Prograph -- Object-Oriented Dataflow Language

> Prograph is a "very high-level, pictorial object-oriented programming environment" that integrates four key trends in computer science: a visual programming language, object orientation, dataflow, and an application-building toolkit. (Wu, 1991, p. 71)

Prograph is a commercial product developed by The Gunakara Sun Systems (TGSS) of Halifax, Nova Scotia, Canada. The ideas behind Prograph are discussed in (TGSS, Tutorial, 1990, chpt. 4) and have been reviewed in the *Journal of Object-Oriented Programming* (Wu, 1991). Both of these references provide detailed information on the Prograph language and its strengths and weaknesses. Here, we discuss only the basics

of Prograph program construction in order to provide the reader with some idea of how our system has been implemented.

Prograph was chosen as our implementation language for several reasons. First of all, its visual dataflow structure is very similar to the approach taken for DFQL. This similarity helped to stimulate our development of DFQL. Also, the ability provided by Prograph to take advantage of the Macintosh visual interface, greatly aided in the development of the DFQL user interface. The fact that Prograph is object-oriented allowed the use of many powerful features of the object-oriented paradigm which also greatly improved the modularity and ability to upgrade and maintain the program code.

The subsequent discussion assumes some knowledge of the ideas of object-oriented programming. The following descriptions should provide enough information to follow the examples in the text and in Appendix C. For further information on the Prograph language the tutorial and reference manuals for the Prograph language (TGSS, 1990) should be consulted.

a. Prograph Code

A simple example of actual Prograph visual dataflow code was given earlier as Figure 6. Dataflow program fragments, such as the one shown in that example, form the methods of the object-oriented paradigm. These methods are grouped into classes, ultimately making up a complete Prograph program. All DFQL classes, their attributes, and their high level methods (along with a brief explanation of Prograph symbology) are included in this thesis as Appendix C.

Prograph provides many primitive operations that are used to construct methods. An example of one of these primitive operations is "show" which prints its input on the screen. Further examples of primitives are the arithmetic operations such as "+" or "-" or trigonometric functions such as "sin" or "cos". Primitive operations are provided by Prograph in the following basic categories: Application, Bit, Data, File, Graphics, Instances, Interpreter Control, I/O, Lists, Logical/Relational, Math, Memory, Strings, System, Text, and Type. These primitives are not methods as defined by the object-oriented model since they do not belong to any class. They are just the Prograph basic operations similar to the operations such as "+" and "-" provided in other object-oriented programming languages such as C++.

Prograph's only built-in complex data structure is the list. The programmer can construct any complex data structure he desires by establishing a class for that purpose, however the level of support inherently provided by the language is at the list level. Therefore, in our DFQL interpreter there are many occurrences of list operations and list data structures. The Prograph primitives for manipulating lists are very powerful and comprehensive. Many of the primitives are reminiscent of LISP list operations. Items can be added, deleted, and inserted at any point in the list. The list can be indexed into by any attribute of the list. A list may contain any Prograph object from a simple data type such as a string, to a more complex type such as another list, to the most complex data object that the user has defined in his application. Also, the objects in a given list do not need to even be of the same type. This supports the idea of using lists to easily implement complex data structures. All list manipulation in Prograph is

done without the use of pointers. This is made possible by providing primitives to index into a given list based on the lists attributes. Primitives are also provided to construct (pack) and disassemble (unpack) lists, again all without the use of pointers.

Another unique aspect of Prograph code is the control structures provided. We have discussed the token model of dataflow programming previously. While Prograph operates on the token model principle, it provides the user with the ability to alter the sequence of program execution. This is especially important when executing operations that have side effects. An example is changing the color of the pen before drawing a figure on the display. The programmer wants to ensure that the color is changed *before* drawing onto the screen. In Prograph this type of operation is specified through the use of *"synchros"* which impose a sequential *order* of execution on operations that otherwise would not be deterministically scheduled. The previous example is shown as Figure 46. The synchro connection between the **ForeColor** method to the **drawitem**

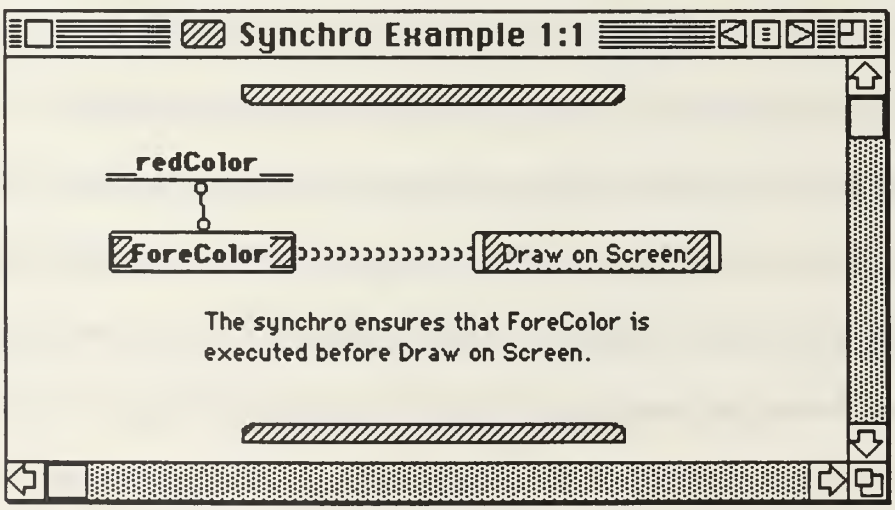


Figure 46. Specifying Order of Execution

method ensures that the color is changed before **drawitem** is executed. If this synchro was not provided either operation could be performed first since neither depends on the other for any input data.

Another type of control structure is required to implement decision making within a Prograph program. This type of capability is provided in most common programming languages as the "if-then-else" and "case" statements. In Prograph, a decision can be made based on any method or primitive that returns a boolean response. Figure 47 shows a method with three cases that will print a message stating whether the input value is less than, equal to, or greater than three. The first case (indicated by the 1:3 in the title bar) of this method tests whether the input value is less than three. The X in the box connected to the comparison primitive ("<") means to go to the next case if the condition is false. Conversely a ✓ in the box means to go to the next case if the condition is true. Thus, in the second case, we check for the number being greater than three, if so we go to the next case. Obviously, the order in which the cases are defined is extremely important. There is no practical limit to the number of cases that can be defined for a method. Defining multiple cases in Prograph allows the coding of the case statement type used in other languages.

Another form of control structure is provided to allow iteration. Iteration may be performed over primitives or methods. There are two basic kinds of iteration implemented--iteration over the elements of a list and simple iteration. To perform the same action on each individual element of a list, a "(...)" notation is used to replace the normal input symbol on the method or primitive. The example in Figure 48 shows

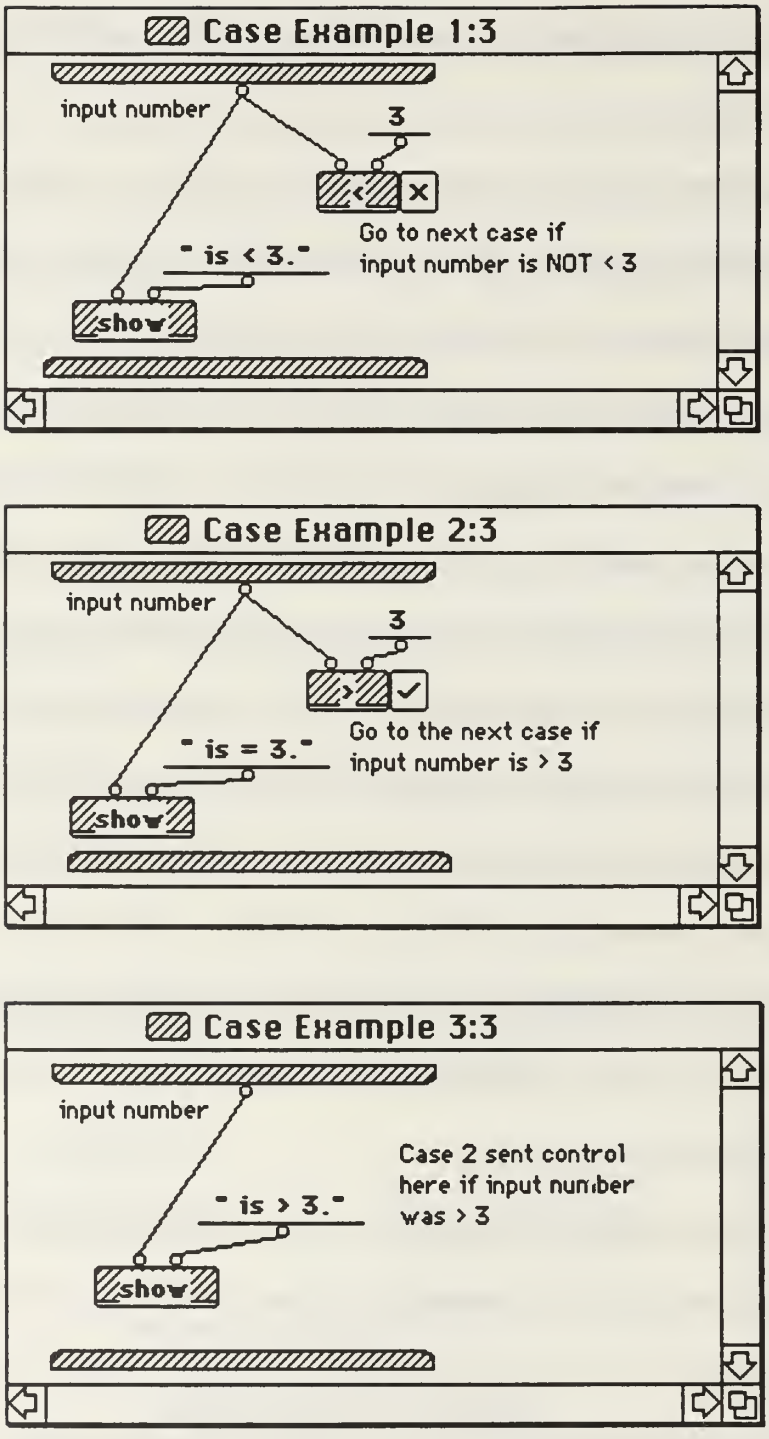


Figure 47. Prograph Case Structure

iteration over a list where 10 will be added to each item in the list. (Since there is no typing enforced in lists it is up to the user to ensure that adding 10 to each of the items makes sense.) Having "(...)" on the output of the "+" operation means that the result will be formed into a list also. List iteration stops when each of the elements in the incoming list has been processed.

Simple iteration is indicated by an arrow linking the output and input of a method as shown in the left side of Figure 49. The internals of the method being iterated are shown on the right side of Figure 49. On simple iteration a condition must be provided in order to stop the iteration. In this example, the condition is specified by using a ✓ with a bar below it attached to a comparison with the number 10. This means to stop the iteration when the condition is true, but to also to allow the current cycle of the iteration to complete. The value of 10 will be propagated as the output of the iterated method. If the bar was above the ✓, then the iteration is stopped immediately and the

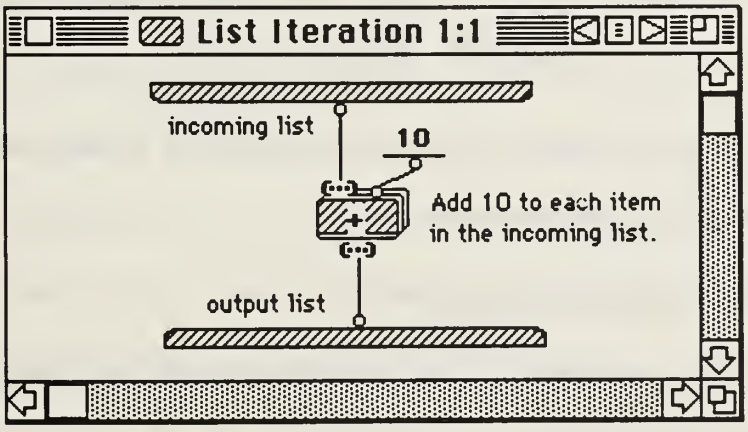


Figure 48. Iteration Over a List

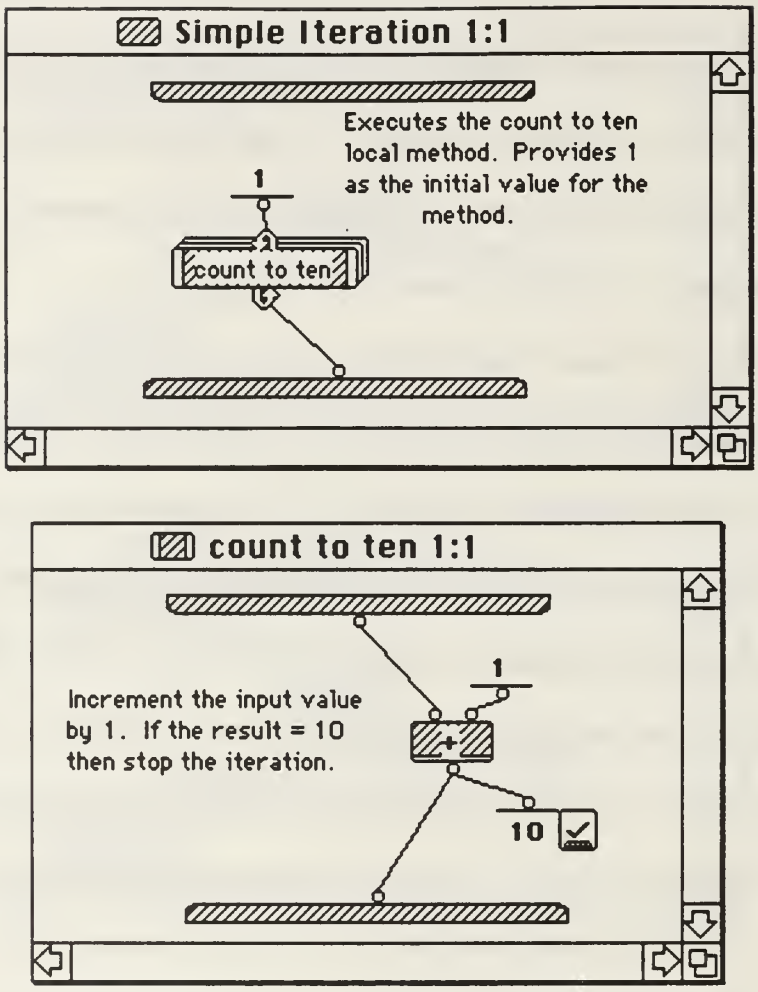


Figure 49. Simple Iteration

value propagated from the output of the iterated method will be the output value of the last completed iteration, in this case nine.

b. Object-Oriented Features

Prograph can be classified as a truly object-oriented language by meeting the definition of object-oriented as implementing objects, classes, and inheritance (Wegner, 1987). All of the object-oriented features are supported entirely visually.

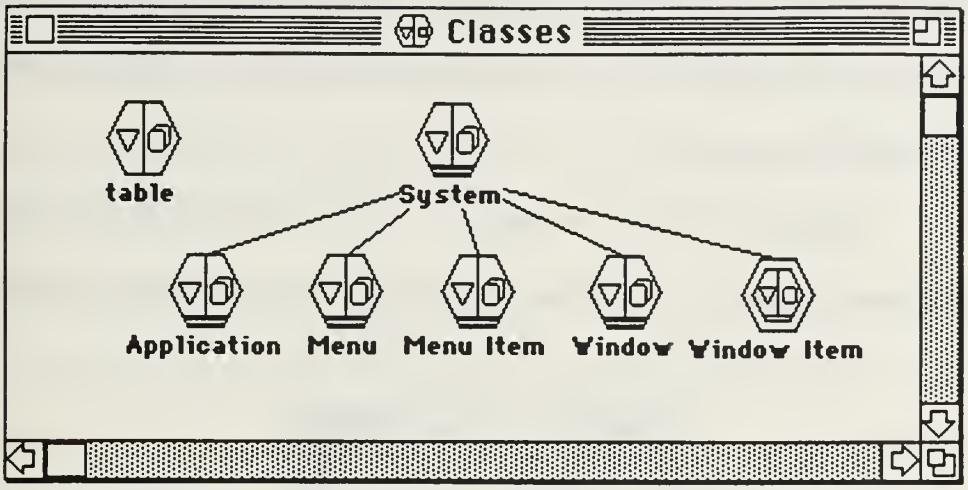


Figure 50. Prograph System Classes

Figure 50 shows the system classes that are provided by Prograph along with the **table** class which is one of the user-defined classes in the DFQL interpreter. Each class is represented by a hexagonal symbol. The class symbols with the double outline indicate that these classes have descendent (or child) classes that are hidden in this display. Child classes can be hidden and revealed through a menu selection in the Prograph editor. The lines between classes represent inheritance links. The methods and attributes from all ancestor classes are inherited by the child classes. There is no "selective" inheritance in which some of the parent's attributes and methods can be inherited while excluding others. Multiple inheritance is also not supported, so the class hierarchy is represented by a true tree structure. System classes are differentiated from user-defined classes in the class diagram by the double line at the bottom of their hexagon. User-defined classes have only a single bottom line. The triangle symbol on the left side of the class symbol represents the class's attributes and the rectangular symbol on the right side of the class

symbol represents the class's methods. By double-clicking on either the left or right side of the class symbol the programmer can open up separate windows displaying the class's attributes or methods respectively.

The class attributes are listed in a vertical column. Figure 51 shows the attributes of the **table** class. Those attributes represented by the hexagonal shape are class

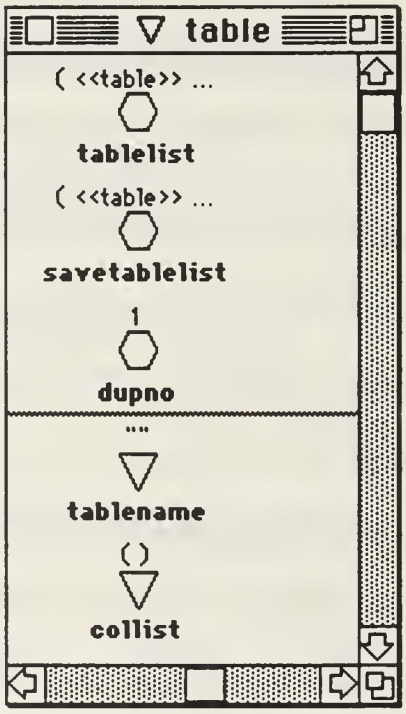


Figure 51. Attribute Window

variables whereas the attributes with the triangular shape are instance variables. If this object had inherited any variables from an ancestor, the inherited variables would be indicated by an arrow imposed on top of the appropriate variable symbol. (For examples of inherited attributes see Appendix C.) Class variables in Prograph are directly

accessible by any instance of the class. Class variables can also be indirectly addressed from anywhere in the application by specifying the class that they belong to and requesting their information. Instance variables are accessible only by the particular instance that "owns" them.

Methods are handled slightly differently than attributes. This is primarily due to not needing an entirely different copy of the class's methods for each instance of the class. Figure 52 is the method window for the **table** class. In the method window, inherited

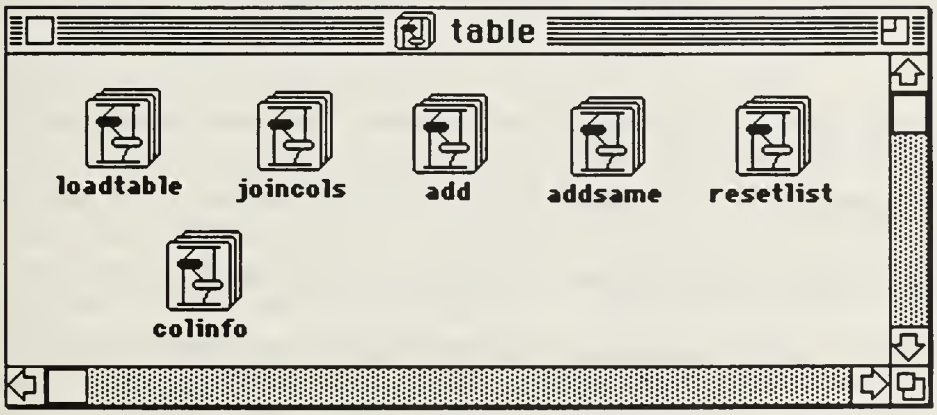


Figure 52. Method Window

methods are not shown. Each of the methods listed in this window may be double-clicked to open an editing window displaying the method's Prograph code. While inherited methods are not show in the class's method window, the instances of that class still have direct access to them. Also, if a method is given the same name as a method in an ancestor class, that method is effectively "overloaded" for the child class. In this case when an instance of the child class calls for the overloaded method it will receive the method in its own class, not the ancestor class, unless specifically requested.

There are three ways of referencing class methods. These are pictured in Figure 53. First, there is *regular* referencing. Regular referencing is indicated by

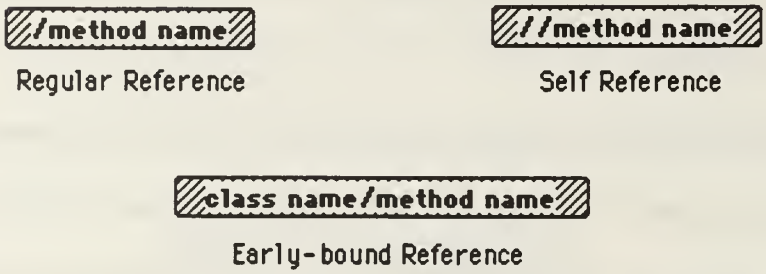


Figure 53. Method Referencing

placing a single slash (/) in front of the method name. This means that the method occurs in the class hierarchy of the object flowing into the method. The second type of referencing is called *self* referencing. This type of referencing is indicated by preceding the method name with two slashes (//). Self referencing means that the method to execute occurs in the same class hierarchy as the current method. The third form of referencing is *early-bound* referencing. This is indicated by prepending the method name with the class name where the method is to be found and a single slash (*classname/*). There is one additional for of method reference that does not fall into the same category as any of the three above. This is the *global* or *universal* form of reference.[11] In Prograph, universal methods can be created that can be accessed by any object simply by specifying the

[11]Terminology is taken from (Wu, 1991). In place of *regular*, *self*, *early-bound*, and *global*, the Prograph manuals use the terms *data-determined*, *context-determined*, *explicit*, and *universal*, respectively.

method name only. This falls somewhat outside of the object-oriented paradigm and is included for the convenience of the programmer.

2. DFQL Implementation Strategy

The block structure of the implementation of DFQL is shown in Figure 54. The DFQL query is entered at the user interface level. The visual query is then stored into a graph structure of database objects (**adbobj** class) which includes the text objects and operator objects. The database objects are then converted into SQL as intermediate code. The SQL code is executed on a backend DBMS (ORACLE in this case), and the results returned from the backend DBMS are displayed for the user.

This implementation strategy was chosen for several reasons. First of all, it separates the actual user interface from the graph processing portion of the DFQL engine. This is designed to allow for easy modification or replacement of the user interface without restructuring the rest of the program. There is an obvious need for a class (**adbobj**) in which to store the graph (ie. nodes and arcs) representation of the query for processing. The generation of SQL as an intermediate code to be run on a separate backend DBMS was chosen as a matter of both portability and expediency. The first factor that influenced this decision was the requirement to run DFQL *on top of* an existing relational DBMS. This requirement was natural since DFQL has been developed as an interface to the relational database. The idea of DFQL is in the interface provided, not in the backend support of the DBMS; there was no need to reinvent a backend to

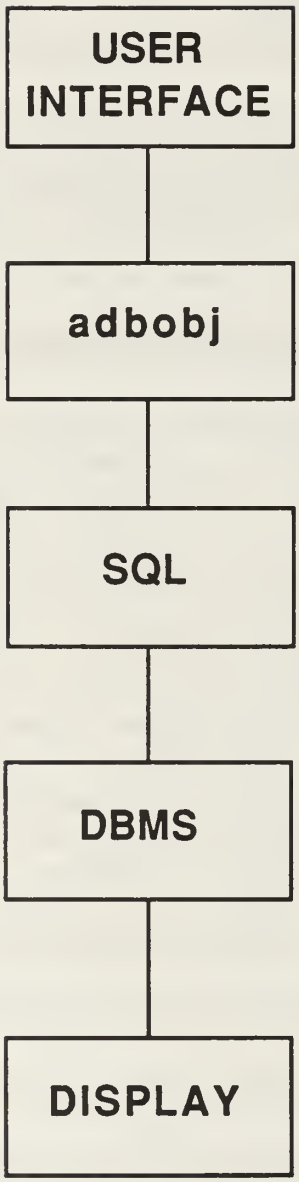


Figure 54. Block Structure of DFQL Interpreter

implement the new query language. The second factor in the decision to use SQL as intermediate code was that in the early development of DFQL it was not known what backend database would be used for implementation; since most relational databases provide an SQL interface, we chose SQL as a common denominator for backend DBMS support. The query is executed as one transaction on the backend DBMS for implementation reasons. The API's required to communicate directly with the backend database for each separate database operator proved to be to onerous to implement for this version of DFQL. Executing the query in "batch mode" on the backend DBMS was orders of magnitude easier to implement and still achieves the goal of linking DFQL to an existing DBMS.

In this implementation of DFQL, the *display* of results was viewed as less important than the generation of the query. Because of this, the display support consists primarily of the ability to produce results from the query that are editable by the user with the normal Macintosh editing facilities. We will now discuss each portion of the implementation structure in more detail.

a. User Interface to Stored Query Graph

The transformation that occurs between the user interface and the stored query graph is shown in Figure 55. The DFQL objects shown on the screen are represented by the **gdbobj** (for **g**raphical **d**ata**b**ase **obj**ect) class. This class and its children have attributes for information that is used exclusively for the display of the DFQL objects. Such information as screen position, is not necessary for the execution

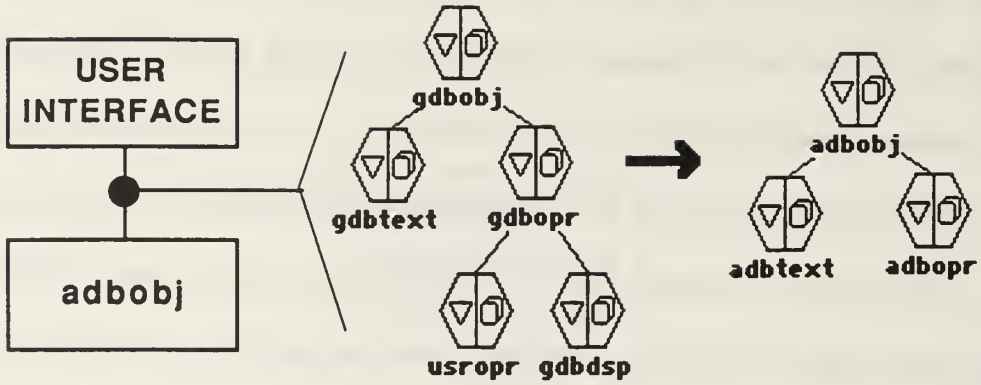


Figure 55. Interface to Object Representation

of the query. Also, the class **usropr** instances of user-defined operators since they are drawn as single objects in the DFQL drawing area. All of the **gdbobj** information about the status of the query currently in the drawing area of the DB INTERFACE window is maintained in a **gdbobj** class variable, **gdbobjlist**. **Gdbobjlist** contains a list of all the DFQL objects in the current query and the connections between them.

The first step in the execution of the query is to ensure that all of the required connections have been made in the query graph. The method **dbops/checkgraph** checks to ensure that all input and output nodes are connected.[12] If this basic criteria is not satisfied, query execution is halted and an error message is displayed. Otherwise, query processing continues by converting the **gdbobj** objects in the **gdbobjlist** into **adbobj** (**a database object**) and placing them in the **adbobjlist**. This involves stripping out all of the display specific information and also coding all **usropr** objects into their

[12]A single output node may remain unconnected for use with the "**Display Last**" feature (p. 94).

constituent text and primitive operator objects. When this process is completed, **adbobjlist** contains the complete query graph for the user's DFQL query in its simplest form of only text and primitive operator objects. The query, as stored in **adbobjlist**, can now be converted into SQL. By keeping the visual objects separate from the **adbobjlist**, graph we gain flexibility in the implementation of the user interface. All that is required from the user interface is that it provide the information required to build the **adbobjlist**.

a. Query Graph to SQL

The conversion of the **adbobjlist** into SQL is performed in accordance with the token model of dataflow programming discussed earlier. This conversion is represented in our block diagram as shown in Figure 56. The program follows dataflows

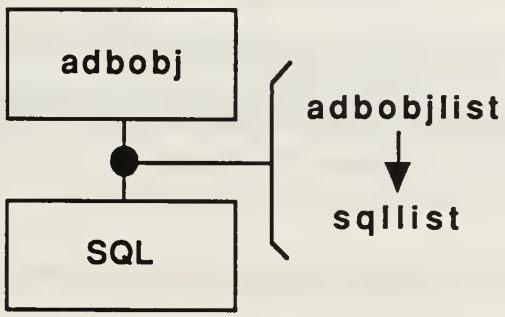


Figure 56. Graph to SQL

from operator to operator, and the operators are "fired" when their shortage counts equal zero. The actual implementation of this algorithm is made in the dbops/doallops method included as Figure 57. This method uses the **find-instance** Prograph primitive to scan **adbobjlist** for objects with shortage counts (**dependnum**) of zero. When such an object

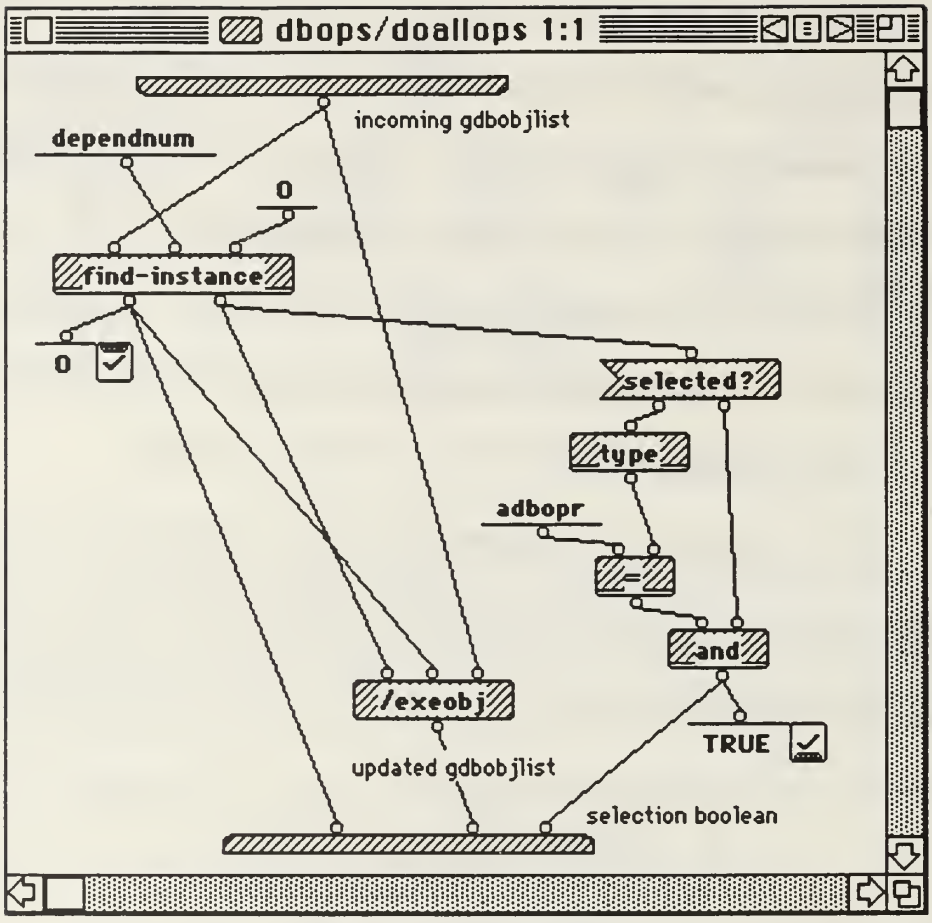
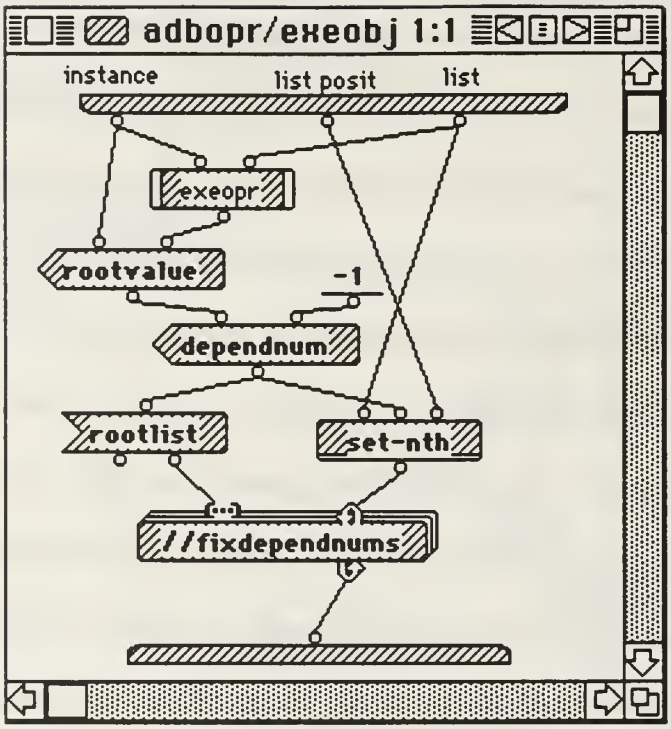


Figure 57. Doallops

is found it is executed by its particular **/exeobj** method. The text objects (**adbtext** class) and the operator objects (**adbopr** class) each have their own different **exeobj** method. When an object is executed it produces a result and updates the shortage counts of the other objects that depend on it. **Adbtext/exeobj** executes the text objects of the query by simply passing on the text value to the connected operators and adjusting shortage counts as necessary. Based on which DFQL operator it receives, **adbopr/exeobj** executes another method to generate the appropriate SQL for the operator. The **dbops/doallops** method is iterated until there are no remaining objects to execute, or until it reaches an

operator that was selected by the user as a stopping point for return of a partial query result. These two stopping conditions for **dbops/doallops** are implemented by the two "value matching" primitives (with the ✓) on the far left and right of the method. The match on the left stops iteration *immediately* when there are no more operators to execute; **/exeobj** *will not* be executed. The match on the right stops iteration following the completion of the current iteration; the operator that was selected will be executed.

The actual SQL code is generated by the execution of each of the individual DFQL operators. The methods that generate the SQL reside in the **dbops** class. They are executed by **adbopr/exeobj** by making use of Prograph's "injection" method. **Adbopr/exeobj** and its local method **exeopr** are shown in Figure 58. A "local method" can be thought of as a subroutine that is visible only to the method it is defined in. Local methods are used to encapsulate portions of their parent methods. The injection construct is shown in **exeopr** as the last object in the dataflow before the output bar. Injection allows the name of the method to be executed to be passed into the method as an argument. That is what the connector into the method with no name indicates in Figure 58. The actual method that is executed in the **dbops** class is one of a group of "**l**" methods. These methods such as **lselect**, **lproject**, etc. take as input a list of arguments for the actual DFQL operator and break the list down into individual elements. These elements are then passed on to the actual method such as **dbops/select** or **dbops/project**. The "**l**" methods are used in order to keep the actual **dbops** methods for query execution isomorphic in structure to their DFQL counterparts. By unpacking the argument lists outside of the execution method the correct number of input nodes is maintained on the



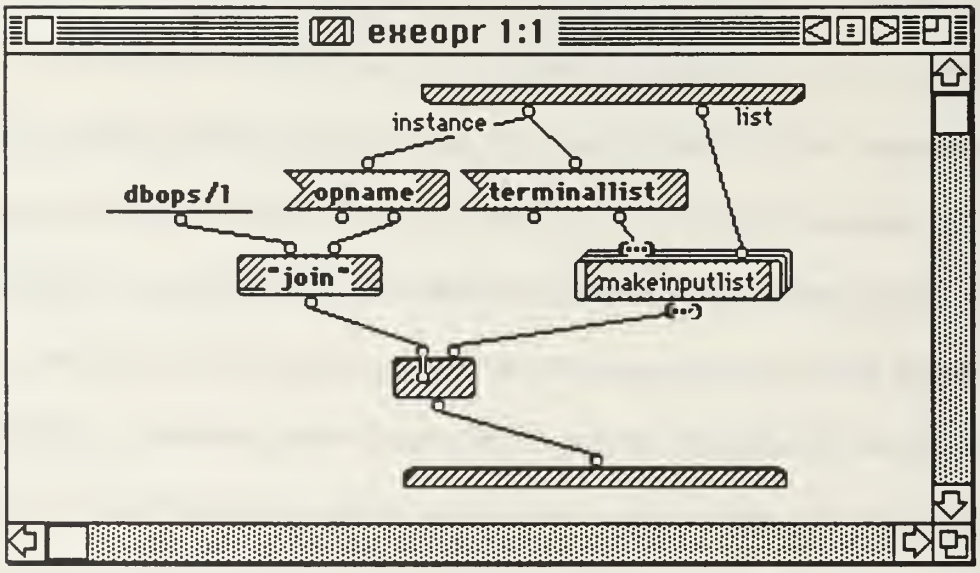


Figure 58. Adbopr/exeobj

actual **dbops** method used for execution. For example the DFQL **join** operator requires three input arguments; by unpacking the argument list using **dbops/lselect**, the **dbops/select** method also has three inputs. The appropriate **dbops** method for each DFQL operator then generates the required SQL query and stores it in a **dbops** class variable called **sqllist**. As an example, **dbops/project** is shown as Figure 59. **Dbops/project** expects two inputs, an incoming relation and an attribute list. The input arguments are inserted into appropriate positions to form a syntactically correct SQL statement which is added to the **sqllist** class variable. All of the other DFQL operators are converted in a similar fashion, although some are significantly more complicated to implement than **project**.

The output of each DFQL operator is required to be a relation. In our SQL system this requirement is met by creating a view as the result of each DFQL operation. Hence the first line added to **sqllist** by **dbops/project** in Figure 59 is "Create view *tempname* as" which is then followed by the constructed SQL statement. The reason for using views to accomplish our goal is twofold. First of all, defining a view does not require that the backend DBMS build a table to represent that view; it may just maintain the rules that define the view. Secondly, in an ideal situation, the backend DBMS should be able to optimize a query that is based on several view definitions by combining their conditions into a single large condition and then doing one optimized data retrieval. At any rate, a view creation (or table creation) is theoretically required for each operator in the query to allow the return of partial results to the user. At the end of the query all of

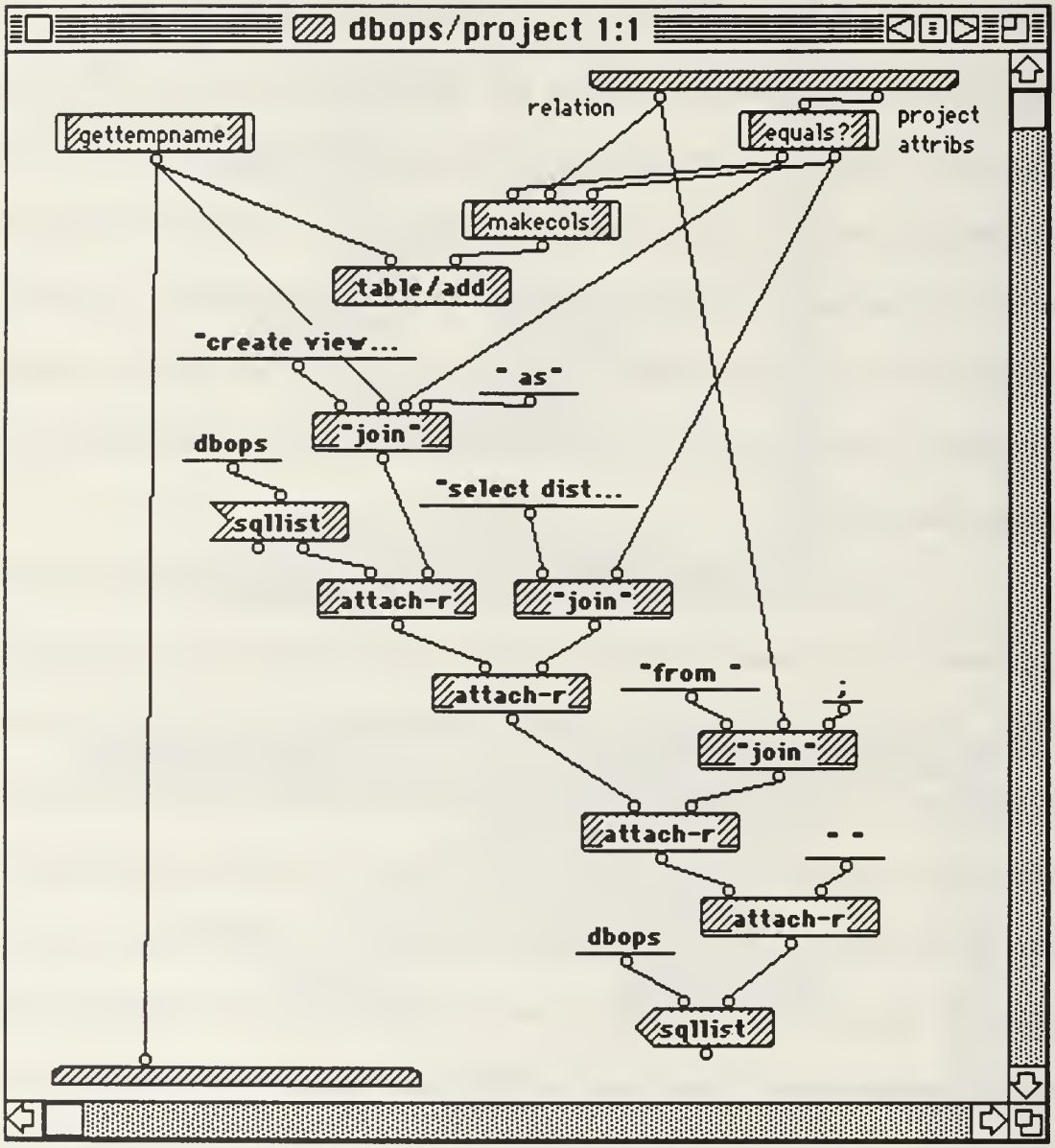


Figure 59. Dbops/project

the temporary views are dropped from the database so as not to clutter it with unnecessary items.

b. SQL to Query Display

Once the query has been translated into SQL it is almost ready to be sent to the backend database as shown in Figure 60. First, the **sqllist** is written out to a file

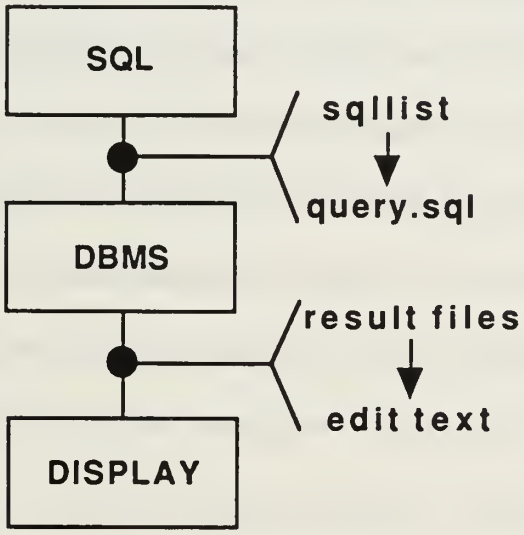


Figure 60. SQL to Result

("query.sql") in ASCII format so it can be read by our backend DBMS SQL interpreter. This is done by the **backend/runoracle** method. Once "query.sql" is written, the ORACLE SQL interpreter, ORACLE*SHELL, is run on the file. During this time, actual control of the computer is transferred to ORACLE*SHELL. ORACLE*SHELL executes the SQL statements in the file and places its output in the file "query.sql.LST". Each of the display operators in the DFQL query will have its own output file generated also.

These output files will be named "spool*n*.LST" where *n* is a unique number associated with a given display operator.

When ORACLE*SHELL releases control of the computer back to the DFQL interpreter, all that remains to be done is the processing of the result file. **backend/loadwinsql** opens the result files and formats them depending on whether or not the user selected the **Show SQL** option or not. If **Show SQL** was not selected, only the output generated by the display operators is retrieved. If **Show SQL** was selected, then all of the information produced during the ORACLE*SHELL run is also retained for display. The results of the query are sent to an editable text object in the Query Results window. By using an editable text output, the user can manipulate the results as desired using the Macintosh cut and paste editor. In this manner the results can also be passed to other applications by exporting them using the Macintosh clipboard.

1. Goals of the DFQL Interpreter Class Structure

Some of the goals of the class structure established for the DFQL interpreter have already been touched on. For example, the separation of the **gdbobj** class from the **adbobj** class was done to allow the user interface objects to be completely divorced from the DFQL query graph processing engine. Another goal in the design was the ability to run DFQL on top of various DBMS's by changing only those methods that directly interface with the backend. Although some DBMS's support different "flavors" of SQL, the DFQL intermediate SQL is primarily based on the ANSI standard for SQL. This should make the SQL generation phase mostly portable. The only methods that actually need to be changed to port DFQL, as it currently exists, to another backend DBMS are

those that are based entirely on the specifics of the ORACLE product. These are grouped together in the **backend** class. DFQL has also been interfaced to an ORACLE DBMS running on a DEC MicroVAX using DECNET over an Ethernet network. This ability shows that DFQL is not simply a Macintosh application but can be used as an interface to other backend databases.

The purported benefits of the object-oriented programming approach were definitely realized in this project. By correctly creating class structures the unrelated parts of the application were kept separate. This separation greatly aided the ability to modify the program. Extensibility has also been greatly enhanced by placing all related methods in single classes. The use of inheritance was of paramount importance in setting up the relationships between the various types of DFQL objects. Prograph's implementation of the object-oriented paradigm definitely influenced the way in which the project was both structured and implemented.

IV. ANALYSIS OF DFQL

A. HUMAN FACTORS ANALYSIS OF QUERY LANGUAGES

Ease-of-use of query languages has been of interest for some time. As more and more information is placed in computer databases that people rely on for making important decisions, the accessibility of that data becomes of increasing importance. No matter how good the data is that an organization may have collected, it is of no value if it cannot be disseminated. Although usability of the DBMS interface is clearly important, it is difficult to quantitatively measure. There have been several approaches to the problem of measuring usability, dating back to the 1970's. While there has been no simple way found to test for usability, contributions have been made in this area that provide some guidelines for evaluation of query languages. In this section we will discuss some of these ideas with emphasis on portions of the previous work that are significant to an analysis of the ease-of-use of DFQL.

1. Testing for Ease-of-Use

Measurement of ease-of-use of query languages is an extension of the field of *human factors analysis*. Human factors analysis is an area which falls under the academic discipline of experimental psychology. The general approach for measuring ease-of-use is divided into three main steps:

- define precisely what is to be measured

- develop a task for users to perform to support the desired measurement
- record the relevant parameters of user performance

The great difficulty lies in applying these steps to activities that involve not only physical and perceptual activities but also cognitive activities such as learning, understanding, and remembering. (Reisner, 1981, pp. 15-16)

Not only is it difficult to construct experiments to measure the above criteria, it is also difficult to interpret the results. Many problems in the interpretation of results lie in the ability to credit a certain factor for the result observed. Often several factors may influence the same result in a given experiment. If this is the case, the experimenter must attempt to determine which is the overriding factor or redesign the experiment to achieve a finer level of granularity in separating the factors involved. In many cases where cognitive issues are involved it is very difficult to isolate all the factors that could affect the experimental results. For example, if learning of a given query language appears to be less than satisfactory as measured by some given criteria, this could mean that the language is "difficult to learn" or it might mean that the method of teaching is not satisfactory.

There are problems both in analyzing a single language for some sense of "absolute" ease-of-use and also in comparing two different languages for "relative" ease-of-use. Although it may seem that it should be simpler to compare two languages than to come up with an "absolute" ease-of-use metric, there are many cautions that are required to avoid drawing incorrect or meaningless conclusions from the comparison

approach. Even with well defined measurement criteria there are often subjective values involved in a comparison. For example, if query language one allows queries to be written twice as fast, on average, as query language two, but query language two produces, on average, 25 percent more *correct* queries, which language should be classified as *easier to use*? This is purely a subjective evaluation. If a person can write twice as many queries then he could more than account for the 25 percent error ratio by the ability to rewrite each query. The preceding statement may or may not be true, but it infers how difficult it is to try and pinpoint usability when left to subjective criteria.

There is also a substantial difference between determining that there is a difficulty with a certain facet of a query language and determining exactly what causes the difficulty or what must be done to fix the difficulty. In fact, studies to determine what is causing a given problem in a language probably should be separated from those trying to determine if a problem exists. This is because there is a different set of criteria required to be measured for each of these tasks, and the experiments required to collect these criteria may be mutually exclusive due to the amount of bias that the method of testing may cause.

2. Applicable Results of Previous Human Factors Studies

The most interesting previously recorded results concerning the ease-of-use issues of database query languages are those that deal with the *procedural* versus *declarative* type of query specification. As noted previously, Codd uses the procedural relational algebra approach for introducing the operations of the relational model in part because "upon first encounter, that approach appears easier to understand;" (Codd, 1990,

p. 62). The performance of procedural and declarative query languages for various different types of queries has been explored. The conclusion drawn by Welty and Stemple from their study is that a more procedural language shows an advantage in the formulation of more difficult queries (Welty, 1981, p. 640). The differences in use of the procedural and non-procedural approaches most likely stem from psychological foundations. In effect, how do humans think when composing a query? If the query passes a certain level of logical complexity, does the human brain naturally break up the query into easier to solve subqueries and then combine these to form the result? Another question is "Could composition be easier with procedural languages but comprehension be easier with specification languages?" (Schneiderman, 1978, p. 428). We concur with Welty and Stemple's conclusion that procedural languages are easier to use for more complex queries. We also believe that any drawbacks to the more procedural approach can be easily mitigated.

Another issue that has been previously explored involves the use of two-dimensional syntax versus linear keyword languages. This issue was discussed somewhat in the section on QBE. It appears that there is some difference in usability of the two types of interfaces based on the actual cognitive abilities of the user. Users who emphasize right brain visual, intuitive thinking have different preferences than users who emphasize left brain verbal, deductive thinking (Schneiderman, 1978, p. 429). In DFQL, the goal is to combine both the right and left brain type of thought processes to gain maximum utility from the language for both types of users.

B. EXPERIMENTAL COMPARISON OF DFQL WITH SQL

In order to come up with some objective measurement of the ease-of-use of DFQL as opposed to that of SQL, we conducted a simple human factors experiment comparing the two languages. The data and additional details of this experiment are included in Appendix B. In this section, we will provide a general assessment of the experiment conducted, and a description of the results of statistical analysis of the recorded data. This experiment is not, and was not intended to be, a *rigorous* comparison of DFQL and SQL. It is only intended to whet the appetite of the readers and researchers regarding the utility of DFQL as a database query language, and as such provides only rough, preliminary investigation results.

1. Assessment of the Experiment

In the experiment 26 subjects were given three queries in English based on the relational schema of Appendix A. The subjects then coded each of the queries, first in DFQL, and then in SQL. Each response was then graded as either "correct" or "incorrect." The composite results were analyzed for statistical significance. We use Reisner's criteria for query language experiment assessment (Reisner, 1981, p. 27) to present the details of our experiment.

a. Subjects

The experiment was conducted on 26 students taking the introductory level database course at U. S. Naval Postgraduate School (NPS) in Monterey, California. Students at NPS are primarily U. S. Military officers; foreign military officers and

Department of Defense civilian employees are also represented. Although the composition of the student body tends to enhance homogeneity, the academic backgrounds of students were quite varied. This is shown by the breakdown of bachelor degree areas presented in Appendix B, Table I. Based on bachelor degree area, subjects were classified as having a "technical" or "non-technical" background. Subjects were also characterized by programming experience. For analysis purposes, subjects with greater than one year of programming experience were classified as "experienced".

b. Teaching Method

The subjects were in the tenth week of a 12 week long introductory database course. They had had over two weeks exposure to relational algebra, relational calculus, and SQL from Instructor A. Instructor B made one 20 minute presentation of DFQL accompanied by a handout describing the DFQL operators and providing some examples of their use. Students also had written course material for the study of SQL. The teaching time for DFQL was limited to one 20 minute session due to constraints of the course. All 26 subjects were in a single section of the database class.

c. Kinds of Tasks

The only kind of activity that was tested was the ability to write queries. This limitation was due to both the constraints involved with the course and the limited goals of the testing.

d. Test Questions

The three test questions were arranged in the perceived order of difficulty. The first question (Q1) involved only selection, projection, and joining to achieve the correct answer. The second question (Q2) required grouping and counting; although this requires only a single operator (**groupcnt**) in DFQL, comprehension is still somewhat more complex than that required for Q1. The third question (Q3) required the use of the universal quantifier and was subjectively viewed as an order of magnitude more difficult than the first two questions. Due to time considerations, only a subset of the functions of either language was tested. The DFQL operators nominally required for the test were: **select**, **project**, **join**, **groupcnt**, and **groupALLsatisfy**. In SQL, the same queries require use of the SELECT... FROM... WHERE clause, employing in Q2 the COUNT(*) aggregate and in Q3 a WHERE NOT EXISTS structure. The questions were designed to require the use of combinations of operators to solve the queries. The latter two questions were asked in areas where the subjective belief was that DFQL is significantly easier to use than SQL.

By providing three levels of difficulty in the questions, it was hoped that there would be a substantive breakout in the results based on difficulty. The intention was also to see if DFQL performed relatively better than SQL in the more difficult queries, as one would expect from the previous work cited comparing procedural and declarative approaches to complex querying.

e. Test Environment

The 30 minute test was conducted at the conclusion of the 20 minute introductory lecture to DFQL. The testing was "open book" with subjects having their class notes on SQL and the brief introductory notes on DFQL from the lecture. Emphasis was placed on accuracy, but the length of the class also posed a time limit on completion. The application that the test questions were taken from is the one that was used to present the introductory DFQL lecture.

Questions were based on the relational schema presented in the lecture to ensure that all subjects had received similar exposure to the particular problem domain. Also, this relieved the subjects from having to assimilate a new schema along with writing the queries in the allotted 30 minute time frame. Since query writing ability and not schema understandability was what was being measured, this seems reasonably appropriate. It is realized that by using the same schema as the one in which DFQL examples had been given in that the results may be slightly biased.

f. Evaluation Method

The criterion evaluated by this experiment was the number of correct queries written by the subjects. The tests were collected and hand-graded[13] by the researchers. Each question was graded as either *essentially correct* or *incorrect.* Essentially correct answers include responses that were either completely correct or contained a minor language or minor operand error. This taxonomy and the following

[13]Some particularly intriguing responses were tested on a DBMS.

definitions were given by Welty and Stemple (Welty, 1981, pp. 635-636). A minor language error is a basically correct solution with a small error that would be found by a reasonably good translator. A minor operand error is a solution with a minor error in its data specification, such as a misspelled column name. However, a transposition of column names (or simply use of the wrong column name) was classified as an incorrect answer because there is no way for the grader, or computer, to determine the subject's intent. It is interesting that in this experiment, most of the responses were either very close to being correct or were completely incorrect.

g. Experimenter Attitude

All attempts were made to eliminate any gross biases from the experiment. Obviously, however, the intent was to show a difference in ease-of-use between DFQL and SQL. Again, our experiment is not purported to be a formal investigation of this issue but merely a preliminary gauge used to attempt to validate some of the researchers' subjective opinions.

2. Experiment Results

A detailed compilation of the experiment data and its breakdown is included as Appendix B. In this section we provide a general discussion of the results derived from the data taken.

The primary measurements were made based on the entire sample population. Subjects were classified as to technical background and programming experience as discussed above, however, these breakdowns did not show any large tendencies not

observed across the entire sample population. The primary metric used was the number of questions answered correctly. This was calculated for each individual question and also for each language as a whole. An *attained level of significance* (p) for each comparison was calculated as discussed in Section D of Appendix B. The attained level of significance basically measures how statistically meaningful the percentage difference in results between DFQL and SQL were for a given comparison. Confidence intervals can also be calculated on each of the comparisons to provide a further feeling for the significance of the reported data.

The "z-test" was used for the statistical analysis of the data. The z-test was chosen due to both its power and its lack of assumption of a given distribution for the data (Matloff, 1988, p. 260). By inspection of the data, and also by the nature of the experiment, we have no outlying data points that would adversely affect the z-test. To use the z-test we must look at the differences (d_i) in number of correct answers between one language and the other for each subject (i) rather than the individual values (X_i) since these values are not independent. Thus, when we establish confidence intervals, for example, the interval we are talking about is the size of the *difference* in percent of correct answers between DFQL and SQL. In analyzing the data we always subtract the SQL percentage from the DFQL percentage; a difference of 20% means that DFQL produced 20% more correct answers than SQL. The experimental results are shown in Figure 61.

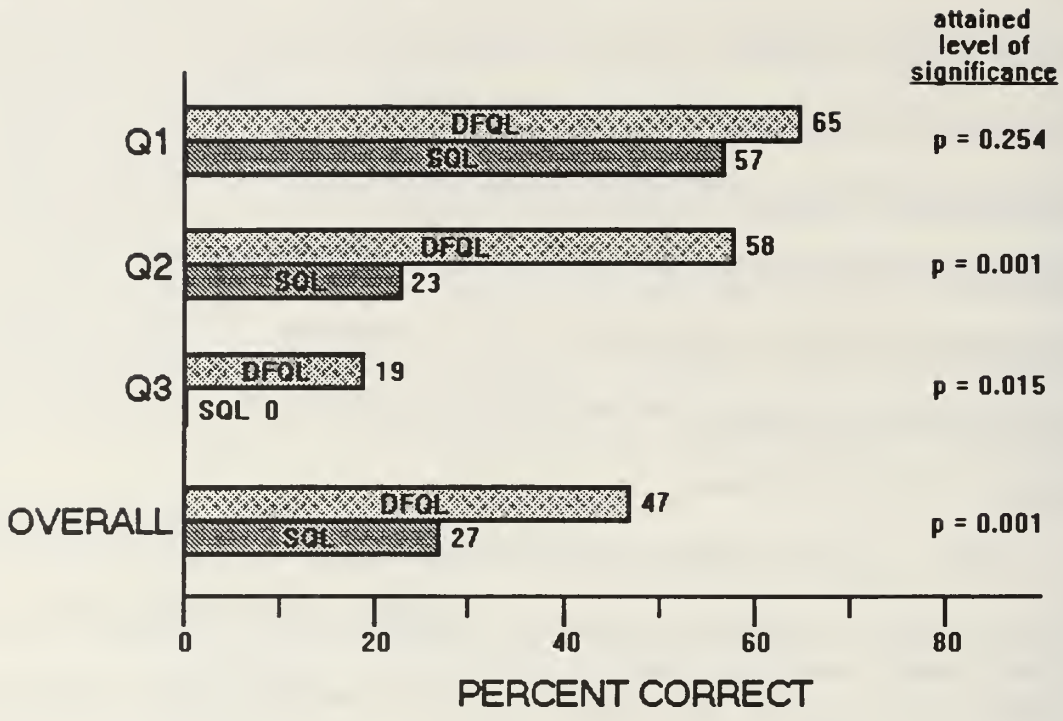


Figure 61. Experiment Results

Figure 61 shows that as the level of difficulty of the query increased a lower percentage of correct answers were made. For the easiest query (Q1) the difference in correct answers between DFQL and SQL was not statistically significant ($p = 0.254$). However, for Q2, Q3, and the overall comparison, significant differences were recorded. The 95% confidence interval ($\alpha = 0.05$) for the overall comparison shows DFQL producing between 8% and 32% more correct queries than SQL. The data seem to indicate, at least for our test criteria, that it is easier to write queries in DFQL than in SQL.

3. Experiment Conclusions

The researchers' perception that queries Q1, Q2, and Q3 were placed in order of increasing difficulty is validated. On the easiest query, there was no significant difference in the number of correct answers achieved by the subjects whether using DFQL or SQL. DFQL produced a significantly higher percentage of correct answers on the more difficult queries. An interesting sidelight to this fact is that of all attempted answers for Q3 there was only one SQL answer that was even "close" to being considered correct; there were five correct DFQL responses to Q3 *and several more* that were "close" to correct. The subjects who formed "close" answers to Q3 in DFQL had a correct DFQL structure for the query, but appeared to mistakenly use either a wrong table or attribute name as an argument in their query.

Both Q2 and Q3 were designed to test areas that are intended to be strengths of DFQL. For example, Q2 requires use of the GROUP BY clause in SQL, whereas it can be coded with a single operator (**groupcnt**) in DFQL. This can help to explain the large difference in performance on Q2 ($\alpha = 0.05$ confidence interval for Q2 shows the difference between DFQL and SQL is [18%, 5%]). Q3 involves universal quantification, which has been previously noted as one of the most difficult concepts to code in SQL. DFQL provides the grouping operators, especially **groupALLsatisfy**, in order to deal with universal quantification. One could say that by testing queries in which DFQL has specific operators provided does not allow a fair comparison of the two languages. However, part of the idea of the experiment was to test areas where DFQL *should* be easier to use because of its operator set. Q2 and Q3 help to validate this claim.

C. ADVANTAGES OF DFQL

DFQL's advantages accrue from the combination of its visual representation, its dataflow structure, and its operator set. The combination of these three characteristics make DFQL unique as a query language and provide it with a unique ability to easily express both simple and complex queries in an intuitive manner. Following is the list of advantages that stem from the DFQL approach.

1. Power

DFQL is relationally complete, and extends the capabilities of first-order predicate logic by the inclusion of grouping operators for both comparison functions and aggregation. The functionality provided directly through the use of the grouping operators was demonstrated in the simple human factors experiment that was conducted and described above. The provided set of primitive operators gives the user the capability of coding practically any desired query.

2. Extensibility

The power of DFQL is enhanced by its ease of extensibility. The user may extend the DFQL language by coding his own user-defined operators from the set of provided primitive operators and also from his own previously created user-defined operators. The user-defined operators are constructed in a manner that fully supports relational operational closure and, once defined, are completely orthogonal with the provided primitive operators. By employing user-defined operators, common operations for any given user can be provided at whatever level of abstraction is desired.

3. Ease-Of-Use

a. Dataflow Representation

Dataflow diagrams were developed to aid in the design of computer programs by providing an easy to use and understand approach to problems that can be functionally defined. DFQL extends this idea to database query languages. A dataflow diagram has the capability, especially when using levels of abstraction (as implemented in DFQL through user-defined operators), to represent even complex problems in an intuitive manner. In DFQL relations as visualized as objects flowing from one operator to another. This ability to view relations as abstract entities directly contributes to the ease-of-use of DFQL. Providing the computer with a dataflow style query graph also enhances its ability to optimize the query for the best possible performance. The human factors experiment conducted provides some data on the ease-of-use of DFQL in writing queries and in subjects' ability to easily pick up the concepts embodied in DFQL. We believe that, at least compared to SQL, DFQL is also easy to read and that the concepts, once learned, are easy to remember.

b. Orthogonality

DFQL provides consistency, predictability and naturalness through use of complete orthogonality of operators. This orthogonality makes the language both syntactically and semantically easier to use. Since relational functional closure is enforced, the user can be assured that the result of **any** operator will be a relation that can then be used as an argument to other operators if desired. Orthogonality is even enforced

in the construction of user-defined operators. The orthogonal features combined with the idea of relations flowing between the operators improve the user's ability to write error-free queries.

c. Incremental Query Formulation and Execution

The ability to form and modify queries incrementally is one of DFQL's most important ease-of-use features. Incremental querying is directly supported by the dataflow structure of DFQL since each dataflow represents an actual relation that can be displayed as a partial result for the user. Intermediate query results can be displayed by use of multiple display statements. Intermediate results may also be obtained by selecting a DFQL operator in the query and then running the query up to that point. Partial results can be returned from any point in a given query and used to help verify or debug the query. Queries can easily be constructed incrementally because of the operational closure of all of the DFQL operators. Since the output of an operator must be a relation, the result of a DFQL operator may always be combined with another DFQL operator to form a more complex query. Subqueries can be coded as user-defined operators if desired to encapsulate the incremental development of complex queries. The combination of all of these features definitely aids the user in the construction of correct queries.

4. Visual Interface

Although the visual interface could be classified along with the other ease-of-use issues, it is so intrinsic to DFQL that it is mentioned separately. The idea of using dataflow diagrams to represent queries has been discussed; the key to the implementation

of DFQL is the ability for the user to easily and interactively build and modify the DFQL dataflow style queries. Allowing the user to interactively manipulate the DFQL query on the computer screen gives a spatial or two-dimensional representation of the query that is lacking from any textual query language. By providing an easy to use interface, DFQL encourages the user to incrementally construct queries, use intermediate results, and in general take advantage of all of the benefits provided by the dataflow approach to query construction. Without a convenient visual user interface none of these benefits would be realized.

D. SHORTCOMINGS OF THE DFQL CONCEPT

We differentiate here between shortcomings in the concept of DFQL and shortcomings in the current implementation. For example, the current implementation of DFQL does not have its own data definition language (DDL) but relies on the underlying relational DBMS for this capability. This is a shortcoming in the present implementation, not a shortcoming in the concept. Problems with the current implementation are discussed in detail in the Future Work section of the following chapter.

1. Interface Problems

One of the most important requirements for a successful implementation of DFQL is the provision of an adequate user interface. The problems we see in this area are typical of problems seen in most visually oriented applications today. The size of the display limits the number of visual objects that can be on the screen at any one time. There may be a corollary here to the "no more than one page of code per procedure" rule

commonly touted in programming language circles. However, by using reasonable sized visual objects an average (say 14") screen becomes cluttered rapidly. In the current implementation of DFQL an attempt is made at mitigating this problem by allowing the drawing area to be scrolled both left and right and up and down. This allows more DFQL code to be "on the system" at the same time but is cumbersome. The user loses the advantage of being able to sit and look at the query as a whole when it must be scrolled back and forth. Another visual problem occurs when there are very many dataflows present in a single query. The dataflow lines invariably become multiply crossed leading to a difficult to follow DFQL diagram. A solution to both of these problems lies in utilizing user-defined operators to their fullest. When the screen becomes too cluttered, encapsulate some portion of it into a user-defined operator. This solution is still only partial, however. Text items for example take up an inordinate amount of space on the screen at any level of abstraction. However, it is difficult to come up with a more compact and convenient way to represent things like complex logical conditions.

2. Language Problems

As a whole, we believe that the DFQL language concept is sound. Dataflow programming is based on ideas that have been in use for some time and are generally accepted as easy to understand and use. We have shown that a working model of DFQL can be interfaced to an existing relational database and that the construction of DFQL queries can be performed after a minimal exposure to the language.

A problem stemming from DFQL's intense visual orientation is the ability to use DFQL in conjunction with other textual computer languages. DFQL queries could

be compiled and inserted into textual programs as functions, however this provides no good way of actually looking at the DFQL code in the context of the program. Such an ability is a common attribute of most *embedded query languages*. A possible solution to this problem would be a textual translation of DFQL which maintains the dataflow paradigm but generates linear text as its interpretation. This would fit in more easily with another textual language but there would still be some impedance mismatch in the *idea* being represented. The text translation of DFQL would still be a *dataflow* oriented object (with all of the implications of non-deterministic execution, etc.) whereas the program it would be embedded into would in most cases be purely procedural/sequential.

There are several other items that could be considered "language problems." These problems though stem from the state of the current implementation and are thus discussed in the Future Work section of the following chapter.

V. CONCLUSIONS

A. REVIEW OF THE RESEARCH

There currently exists a need for an improved query language for the relational model of database management. This new query language is required to allow users to better harness the inherent power of the relational model. In this research we have designed, implemented, and tested a graphical dataflow query language, *DFQL*, to meet this need.

DFQL was first conceived on paper. Actual implementation was then performed using the Prograph visual dataflow programming language on a Macintosh II/ci computer. ORACLE was used for the backend database. DFQL has been run on a local database established on the Macintosh and also on a remote database by access over an Ethernet network. DFQL has proven to be a workable query language with many benefits over the current de facto standard SQL.

B. FUTURE RESEARCH

The development and implementation of DFQL has brought to light several areas where further research and development needs to be done. These areas relate primarily either to implementation enhancement or theoretical investigation.

1. Implementation Enhancement

The current connection of DFQL to the backend database is through the use of file passing and batch execution. A possible improvement in performance could be gained through direct connection of DFQL to the backend DBMS. This could be done by using an *interface class* to provide the connection between DFQL and the DBMS. The interface class would utilize the necessary API's to communicate with whatever backend database was being used. Another approach to direct connection of DFQL to a database would be for the backend database to be written in Prograph, specifically for DFQL. It is possible that a Prograph backend database could further increase performance, but this would place a needless restriction on the implementation. The ability of the DFQL interpreter to run on top of any currently existing database (as long as API's can be provided) is viewed as a very important point. Investigation needs to be done to determine exactly *how much* efficiency would be improved by use of a Prograph backend database before this option is considered.

Further development can be put into the design of the user interface. Several enhancements could be made to the DFQL drawing area. For example, allowing the user to create new DFQL objects simply by clicking the mouse and allowing the entering of text right on the screen would be improvements. The method of partial query execution could be changed to allow the partial query to be executed simply by double clicking the mouse on the output of an operator. Many good user interface ideas can be identified in the design of the Prograph editor, some of these ideas (such as those above) could be incorporated into the DFQL user interface.

Currently, the Info menu is limited to providing the column names of the tables in the current database. Enhancements in this area of the user interface would include the ability to display the schema for the user and possibly even provide as previously coded user-defined operators for the commonly executed joins that represent relationships in the schema. Rather than having to type in the name of the relation desired, the user could pick a mini-icon off the schema diagram that would represent a given relation. This would somewhat alleviate the problem of crowding the DFQL drawing area with text information.

The current implementation of DFQL provides only data retrieval capabilities, thus requiring the user to directly access the backend DBMS for other functions. Another enhancement to DFQL should provide data definition capabilities and also an enlarged scope of query activities to include database updates and deletions. Inclusion of these features will make DFQL a complete database language.

2. Theoretical Investigation

Much work can be done in the area of optimization of the query graph. Since DFQL implements queries as a graph of relatively low level operations, many of these operations should be able to be combined and reordered tc maximize efficiency. The optimization idea harks back to Dadashzadeh's work in translating SQL into a relational algebra graph in order to help with optimization (Dadashzadeh, 1990, p. 308). In the case of DFQL, a query graph is present at the outset, all that remains is to optimize it.

The DFQL primitive operator set can be expanded. Possible candidates for inclusion into the operator set would be relational operations such as inner and outer join.

Also, additional grouping operators such as group containment could be implemented. Further study is needed to determine if these and possibly other operators are of such convenience (or necessity) that they should be provided for the users.

A possible extension to the language would be to allow it to handle relational valued attributes or even objects in its relations. An extension of this type should maintain the ideas of the relational model for which DFQL was designed. In order to expand the language in this manner further consideration will need to be given to the type of backend database that DFQL will be connected to. Handling relationally valued attributes or other objects is not supported by current relational DBMS's. To implement this in DFQL would require DFQL to map these complex structures to a current DBMS or use a new backend DBMS designed specifically to handle these types of constructs. This type of extension would represent a major change in the scope of DFQL.

Lastly, further human factors analysis should be conducted on the DFQL language. The goal of this analysis should be to quantify the ease-of-use of DFQL both in an absolute sense and also in comparison to other currently used query languages. The experiment conducted as part of this thesis was cursory in nature. More in-depth analysis of DFQL's performance to include a variety of environments, subjects, query types, and comparison languages should be done to validate our feelings about DFQL's superiority as a database query language.

C. SUMMARY

DFQL was created in order to provide an improved interface to the relational model of database management. DFQL presents an entirely new way of visualizing database queries. DFQL's dataflow structure and orthogonality greatly aid the user in the composition of complex queries. DFQL allows users the ability to easily extend the language by the creation of user-defined operators. These user-defined operators can then be used to simplify queries by introducing levels of abstraction, effectively hiding detailed query operations. We conducted a simple human factors experiment, in which DFQL compared favorably to SQL for use in query writing.

While there have been other attempts at producing graphical type interfaces to database systems, none embodies the powerful features that have been designed into DFQL. The unique combination of a visual interface, the dataflow programming paradigm, and the relational model make DFQL a superior choice for continued research and implementation.

LIST OF REFERENCES

Abiteboul, S., and Hull, R., "IFO: A Formal Semantic Database Model," *ACM Transactions on Database Systems*, v. 12, pp. 525-565, December 1987.

Andyne Computing Limited, *GQL: Graphical Query Language; GQL/User Demo Guide*, Kingston, Ontario, March 1991.

Angelaccio, M., Catarci, T., and Santucci, G., "QBD*: A Graphical Query Language with Recursion," *IEEE Transactions on Software Engineering*, v. 16, pp. 1150-1163, October 1990.

Apple Computer, Inc.. *Inside Macintosh*, v. 1. Addison-Wesley, 1985.

Beech, D., "New Life for SQL," *Datamation*, v. 35, pp. 29-36, 1 February 1989.

Bryce, D., and Hull, R., "SNAP: A Graphics-based Schema Manager," *Proceedings of the Second IEEE International Conference on Data Engineering*, pp. 151-164, February 1986.

Chamberlin, D. D., and Boyce, R. F., "SEQUEL: A Structured English Query Language," *Proceedings of the ACM--SIGFIDET Workshop*, Ann Arbor, Michigan, May 1974.

Chen, P. P., "The Entity-Relationship Model -- Toward a Unified View of Data," *ACM Transactions on Database Systems*, v. 1, March 1976.

Codd, E. F., "Relational Completeness of Data Base Sublanguages" in *Data Base Systems*, pp. 65-98, Prentice-Hall, 1972.

Codd, E. F., "Fatal Flaws in SQL: Part I," *Datamation*, v. 34, pp. 45-48, 15 August 1988.

Codd, E. F., "Fatal Flaws in SQL: Part II," *Datamation*, v. 34, pp. 71-74, 1 September 1988.

Codd, E. F., *The Relational Model for Database Management: Version 2*, Addison-Wesley, 1990.

Czejdo, B., and others, "A Graphical Data Manipulation Language for an Extended Entity-Relationship Model," *IEEE Computer*, v. 23, pp. 26-36, March 1990.

Dadashzadeh, M., "An Improved Division Operator for Relational Algebra," *Information Systems*, v. 14, pp. 431-437, 1989.

Dadashzadeh, M., and Stemple, D., "Converting SQL queries into relational algebra," *Information & Management*, v. 19, pp. 307-323, December 1990.

Date, C. J., "Where SQL Falls Short," *Datamation*, v. 33, pp. 83-86, 1 May 1987.

Davis, A. L., and Keller, R. M., "Data Flow Program Graphs," *IEEE Computer*, v. 15, pp. 26-41, February 1982.

Elmasri, R., and Navathe, S. B., *Fundamentals of Database Systems*, Benjamin/Cummings, 1989.

IBM Research Report RC 16877 (#73833), *GRAQULA: A Graphical Query Language for Entity-Relationship or Relational Databases*, by Sockut, G. H., and others, 14 March 1991.

Kim, H., Korth, H. F., and Silberschatz, A., "PICASSO: A Graphical Query Language," *Software-Practice and Experience*, v. 18(3), pp. 169-203, March 1988.

Kim, W., "On Optimizing an SQL-like Nested Query," *ACM Transactions On Database Systems*, v. 7, 1982.

Matloff, N. S., *Probability Modeling and Computer Simulation: Applied to Engineering and Computer Science*, PWS-KENT Publishing Co., 1988.

Miyao, J., and others, "Design of a High Level Query Language for End Users," paper presented at the 1986 IEEE Workshop on Languages for Automation, National University of Singapore, Kent Ridge, Singapore, 27-29 August 1986.

Negri, M., Pelagatti, G., and Sbattela, L., "Short Notes: Semantics and Problems of Universal Quantification in SQL," *The Computer Journal*, v. 32, pp. 90, 91, 1989.

Ozsoyoglu, G., and Wang, H., "A Relational Calculus with Set Operators, Its Safety, and Equivalent Graphical Languages," *IEEE Transactions on Software Engineering*, v. 15, pp. 1038-1052, September 1989.

Ozsoyoglu, G., Matos, V., and Ozsoyoglu, Z. "Query Processing Techniques in the Summary-Table-By-Example Database Query Language," *ACM Transactions on Database Systems*, v. 14, pp. 526-573, December 1989.

Reisner, P., "Human Factors Studies of Database Query Languages: A Survey and Assessment," *Computing Surveys*, v. 13, pp. 13-31, March 1981.

Sebesta, R. W., *Concepts of Programming Languages*, Benjamin Cummings, 1989.

Schneiderman, B., "Improving the Human Factors Aspect of Database Interactions," *ACM Transactions on Database Systems*, v. 3, pp. 417-439, December 1978.

Shu, N. C., *Visual Programming*, Van Nostrand Reinhold, 1988.

TGSS (The Gunakara Sun Systems Limited), *PROGRAPH: Tutorial*, second printing, 1990.

TGSS (The Gunakara Sun Systems Limited), *PROGRAPH: Reference*, second printing, 1990.

Washington University Department of Computer Science, WUCS-86-5, *Determinancy of Hierarchical Dataflow Model: A Computation Model for Visual Programming*, Kimura, T. D., March 1986.

Wegner, P., "Dimensions of Object-Based Language Design," *OOPSLA '87 Proceedings*, October 1987, Orlando, Florida; special issue of *SIGPLAN Notices*, v. 22, December 1987, pp. 168-182.

Welty, C., and Stemple, D. W., "Human Factors Comparison of a Procedural and a Nonprocedural Query Language," *ACM Transactions on Database Systems*, v. 6, pp. 626-649, December 1981.

Wong, H. K. T., and Kuo, I., "GUIDE: Graphical User Interface for Database Exploration," *Proceedings of the Eighth International Conference on Very Large Databases*, pp. 22-32, September 1982.

Wu, C. T., "OOP + Visual Dataflow Diagram = Prograph," *Journal of Object Oriented Programming*, pp. 71-75, June 1991.

Yourdon, E., *Modern Structured Analysis*, Prentice-Hall, 1989.

Zloof, M. M., "Query-by-Example: A Data Base Language," *IBM Systems Journal*, v. 16, pp. 324-343, 1977.

APPENDIX A. EXAMPLE DATABASE

The schema and data for the database referenced by the examples in the text is included here. Most of the relationships between the data should be apparent. The intention is to represent a simple university database in which students are enrolled in courses taught by instructors. An Entity-Relationship Diagram of the database is shown in Figure 2 of the thesis (p. 21).

The relational schema is listed below. In the schema representation, keys are underlined.

COURSE(CNO, TITLE, INO)

ENROLL(CNO, SNO, GRADE, TESTSCORE)

INSTRUCTOR(INO, INAME, PAY)

STUDENT(SNO, SNAME, ADDR, PHONE, GPA)

Attribute definitions:

ADDR -- Address
CNO -- Course Number, unique to a single course
GPA -- Grade Point Average
GRADE -- Course Grade ('A', 'B', 'C', etc.)
INAME -- Instructor Name
INO -- Instructor Number, unique to a single instructor
PAY -- Instructor's Pay
PHONE -- Phone Number
SNAME -- Student Name
SNO -- Student Number, unique to a single student
TESTSCORE -- Numerical Grade for an exam in a course
TITLE -- Name of a Course

The example data in the database is listed below.

COURSE	CNO	TITLE	INO
	CS05	COURSE # 5	I1
	CS10	COURSE #10	I2
	CS15	COURSE #15	I3
	CS20	COURSE #20	I2
	CS25	COURSE #25	I3

ENROLL	SNO	CNO	GRADE	TESTSCORE
	S1	CS10	A	92
	S1	CS15	C	72
	S1	CS20	A	93
	S2	CS05	A	98
	S2	CS10	A	95
	S2	CS25	A	90
	S3	CS05	B	85
	S3	CS10	A	91
	S4	CS05	A	93
	S4	CS15	B	83
	S4	CS25	A	94
	S5	CS05	C	70
	S5	CS15	B	82
	S5	CS20	A	94

INSTRUCTOR	INO	INAME	PAY
	I1	INST #1	100000.00
	I2	INST #2	50000.00
	I3	INST #3	47380.78

STUDENT	SNO	SNAME	ADDR	PHONE	GPA
	S1	STU #1	ROOM 1	111-1111	3.85
	S2	STU #2	ROOM 1	111-1111	3.40
	S3	STU #3	ROOM 3	333-3333	3.75
	S4	STU #4	ROOM 3	444-4444	2.85
	S5	STU #5	ROOM 5	555-5555	3.30

APPENDIX B. HUMAN FACTORS EXPERIMENT DATA

Three queries in english were posed to a group of 26 computer science students at Naval Postgraduate School. The subjects were asked to write each query first in DFQL and then in SQL. These queries were based on the relational schema of Appendix A. While all subjects were computer science master's degree students, there was some variation in their background. Each individual was categorized by technical background (based on their bachelor's degree) and programming experience (whether greater than one year).

A. Queries

Q1. List courses (cno) taught by those who earn more than 50K.

Q2. For each instructor, list the number of courses he taught.

Q3. List all instructors (ino) who gave only A's in all the courses they taught.

B. Definition of Technical Background

Bachelor degree areas were classified as technical or non-technical as shown in Table I.

Table I. SUBJECT BACKGROUND

TECHNICAL	NON-TECHNICAL
Applied Science	Accounting
Chemical Engineering	Bible Studies
Chemistry	Business Administration
Computer Science	Journalism
Control Systems	Liberal Arts
Electrical Engineering	Political Science
General Engineering	Production System Mgmt
Mathematics	Zoology
Mechanical	
Natural Science	
Petroleum Geology	
Physical Science	

C. Data

The data that was collected from the experiment is listed in Table II. Subjects are listed numerically from 1 to 26. Individual performance on each query is listed for DFQL and SQL. A "0" indicates an incorrect answer, and a "1" indicates a correct answer. Summary percentages for each question and each language are included at the bottom of the table.

Table II. Collected Data

Subject	Tech	Program > 1yr	DFQL			SQL		
			Q1	Q2	Q3	Q1	Q2	Q3
1	Y	N	1	1	0	1	0	0
2	Y	N	0	0	0	0	0	0
3	N	Y	1	1	0	1	0	0
4	Y	N	1	0	0	1	0	0
5	N	Y	0	0	0	0	0	0
6	Y	Y	1	1	1	1	1	0
7	N	Y	1	0	0	1	0	0
8	Y	N	1	1	0	1	1	0
9	Y	N	0	0	0	1	0	0
10	Y	Y	1	1	1	0	0	0
11	Y	Y	1	1	0	1	1	0
12	Y	Y	1	1	0	1	1	0
13	Y	Y	0	0	0	1	0	0
14	Y	N	1	1	0	1	0	0
15	N	Y	1	1	1	1	0	0
16	N	N	1	0	0	1	0	0
17	Y	N	0	1	1	0	0	0
18	Y	Y	1	1	0	1	1	0
19	Y	N	1	1	0	1	0	0
20	Y	Y	0	0	1	0	0	0
21	Y	Y	0	0	0	0	0	0
22	N	N	0	0	0	0	0	0
23	Y	Y	1	1	0	0	0	0
24	Y	Y	1	0	0	0	0	0
25	N	N	1	1	0	0	1	0
26	N	N	0	1	0	0	0	0
		Percent Correct	**65**	**58**	**19**	**57**	**23**	**0**
			47			**27**		

D. Analysis

The analysis of our data is predicated on the knowledge that our sample size is both small and rather homogeneous. Practically all students at Naval Postgraduate school are either military officers or civilian Department of Defense employees. Homogeneity is increased by using only computer science students for our sample (although at Naval Postgraduate School many computer science students come from dissimilar backgrounds as shown in Table I). The test given was limited to six questions due to time considerations. Within the restrictions implied by the preceding constraints, we have produced statistically significant results.

Confidence intervals and levels of significance were established for the data using the "z-test." Since the same subjects were used to test both DFQL and SQL on the same queries the values are not independent. Because of this the z-test was used on the difference (d_i) between the number of correct answers for DFQL ($X_{i,DFQL}$) and SQL ($X_{i,SQL}$) for each subject (i). The same analysis is done also for each question individually.

The null hypothesis, H_O, for the level of significance testing, is that there was no difference in the average number of correct answers between DFQL and SQL. The alternative hypothesis, H_A, is that DFQL produced a higher average number of correct results than SQL.

E. Equations Used in Analysis

(1) Difference $$d_i = X_{i,DFQL} - X_{i,SQL}$$

(2) Mean Difference $$\bar{d} = \frac{1}{n}\sum_{i=1}^{n} d_i$$

(3) Sample Variance $$S_d^2 = \frac{1}{n-1}\sum_{i=1}^{n} (d_i - \bar{d})^2$$

(4) Confidence Interval $$\left[\bar{d} \pm z\left(\frac{\alpha}{2}\right)\frac{S_d}{\sqrt{n}}\right]$$

(5) Hypotheses $$H_O: \bar{d} = 0$$
$$H_A: \bar{d} > 0$$

(6) Observed Significance Level $$p = 2z^{-1}\left(\frac{\bar{d}\sqrt{n}}{S_d}\right)$$

F. Breakdown by Category

Since the percentage differences between DFQL and SQL for all categories were nearly similar and the number of subjects in individual categories was small (due to small overall sample size), as shown above in Table III, detailed statistical analysis was performed only on the total sample data. In the technical/non-technical category there

Table III. PERCENT CORRECT BY CATEGORY

CATEGORY	NUMBER	% CORRECT	
		DFQL	SQL
Technical	18	50	30
Non-Technical	8	42	21
Experience > 1 yr	14	52	29
Experience ≤ 1 yr	12	41	25
Total Sample	26	47	27

was a difference of approximately 10% in both the DFQL and SQL percentages. In the experience category there was a difference of approximately 10% in the DFQL scores and only 4% in the SQL percentages. While the 4% is not in itself statistically significant, a possible explanation for the discrepancy is that the technical background factor may have been more important than the programming experience factor[14] in the ability to use SQL. There were subjects with technical background in the experience ≤ 1 yr category.

[14]None of the subjects could be classified as *professional* programmers.

This would imply that DFQL was easier to use for the non-technical background people than SQL. There is not enough data to support this statement statistically.

APPENDIX C. DFQL SOURCE CODE

This appendix lists the meanings of some of the more common Prograph programming notations. The DFQL interpreter class hierarchy is included along with the attributes and methods for each class. The top level methods are shown for each class. Methods provided by Prograph as part of the Prograph system classes are not listed.



method input bar



Takes inputs and produces outputs as defined.



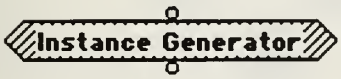
Passes value of Constant as output.



Results in a true or false condition depending on whether input "matches" the MATCH value. This condition determines the effect of the control -- ie. next case, etc.



Pass in a value to set the Persistent.
Output produces current value of Persistent.



Create instance of type named.



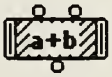
Input: object (for instance variables), class name (for class variables)
Output: left -- passes through input, right value of attribute named in the get



Input: left -- object or class name, right -- new value
Output: Object with new named attribute reset to new value (or class name if a class variable was set.



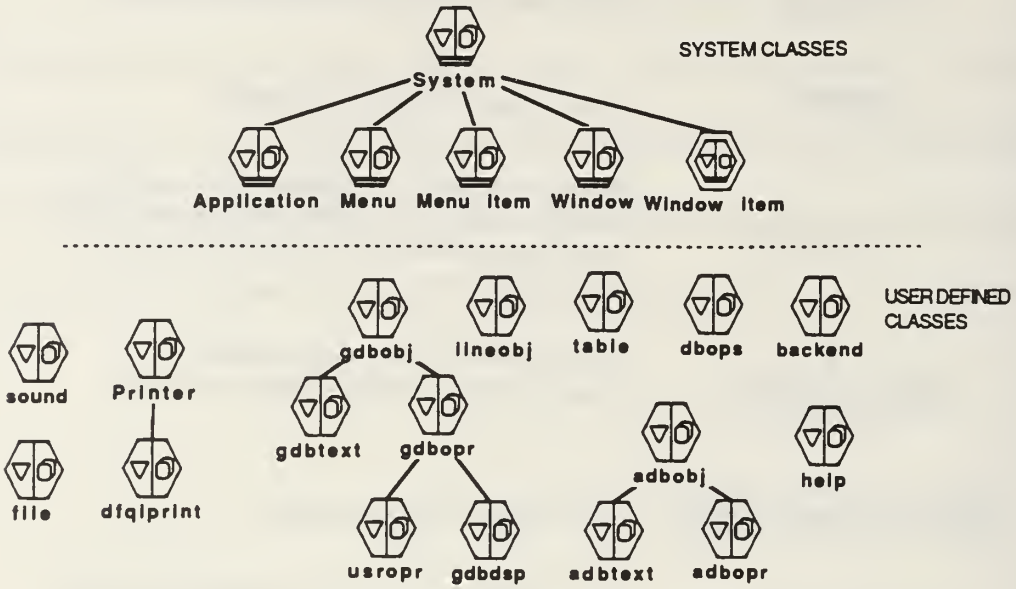
Used to encapsulate operations in a method (like a sub method).



Evaluate method. Pass in variables (which are then named a, b, c, etc.) and use them in the equation producing an output.



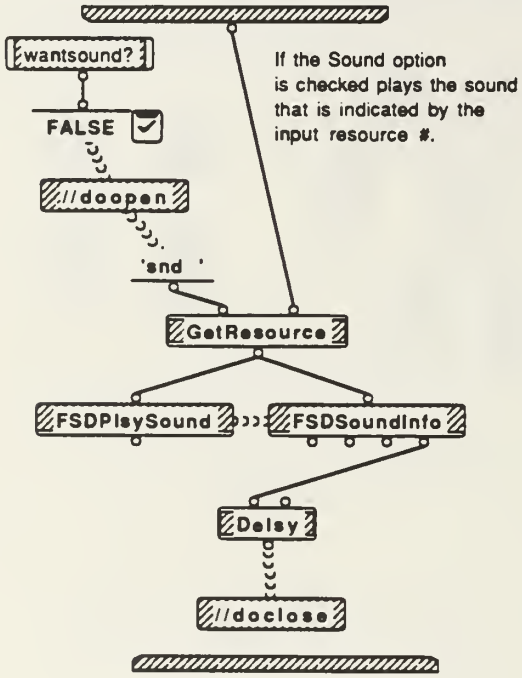
output bar


SYSTEM CLASSES
System
Application
Menu
Menu Item
Window
Window Item
USER DEFINED CLASSES
sound
Printer
file
dfqiprint
gdbobj
lineobj
table
dbops
backend
gdbtext
gdbopr
usropr
gdbdsp
adbobj
help
adbtext
adbopr

sound



sound/play 1:1

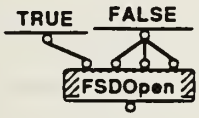


sound/doclose 1:1



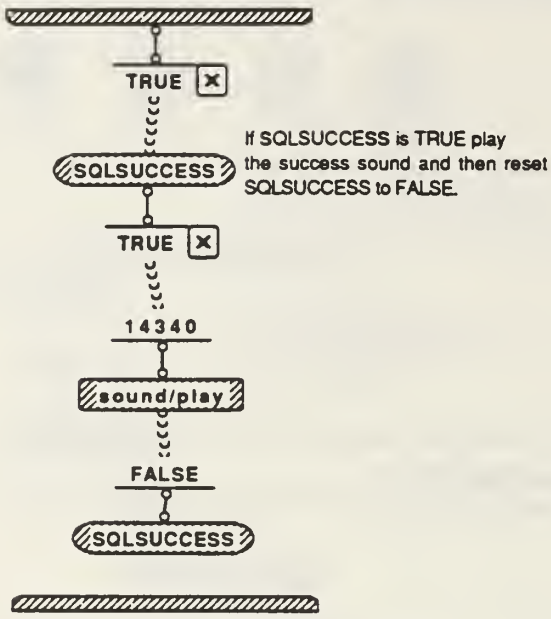
Closes the Farralon Sound Driver

sound/doopen 1:1



Opens the Farralon Sound Driver

sound/succsnd 1:2



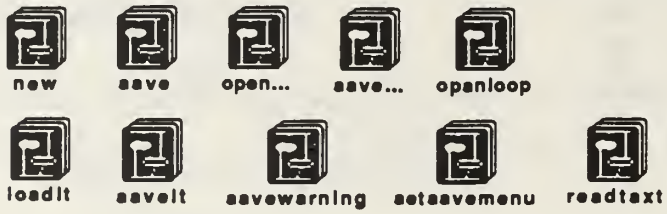
sound/succsnd 2:2



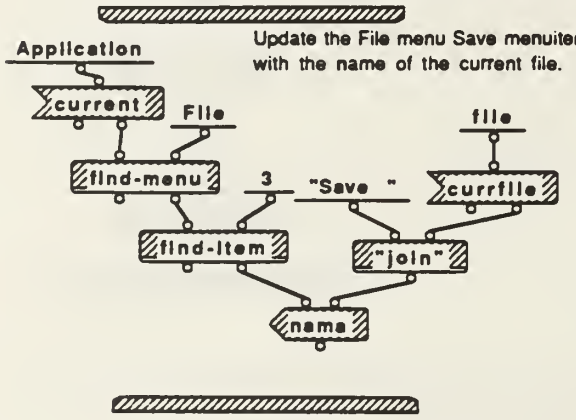


∇ file
"untitled" currfile TRUE changeflag

file

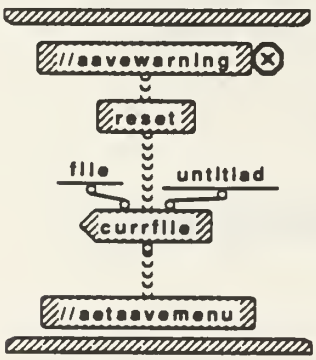


file/setsavemenu 1:1



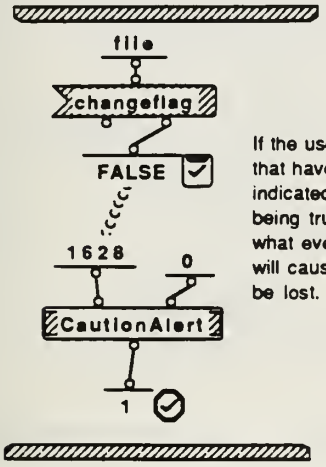
Update the File menu Save menuitem with the name of the current file.

file/new 1:1



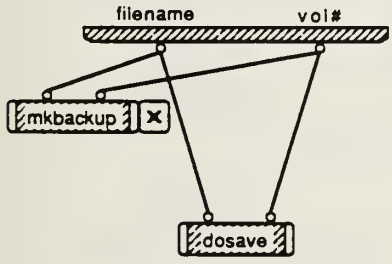
Called in response to the New menu item in the File menu. Resets DFQL and sets current file to "untitled".

file/savewarning 1:1



If the user has made changes that have not been saved as indicated by changeflag being true. Warn him that what ever he is going to do will cause these changes to be lost.

file/saveit 1:2



First backup the file if possible, and then save the data from areclist into filename on vol#.

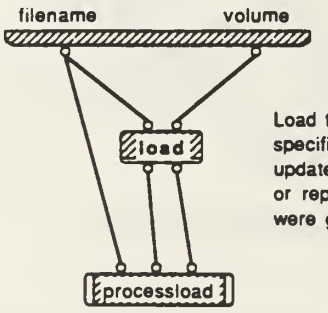
file/saveit 2:2



If the backup operation failed, then don't save the new file because it could write over our previous data without having that data backed up.

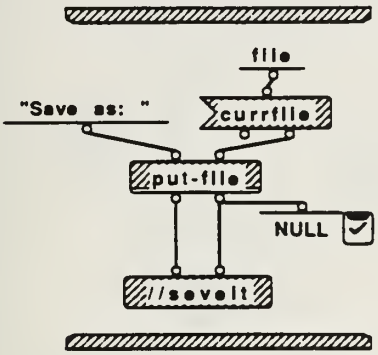
The error messages for failed backup are all contained in the mkbackup local method.

file/loadit 1:1



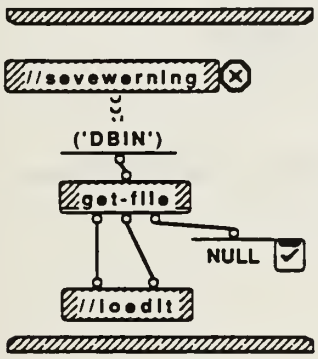
Load the data from the specified file. Processload updates the gdbobjlist and window or reports on any errors that were generated by the load.

file/save... 1:1



Save the data into
the file entered by the user through
the put-file dialog.

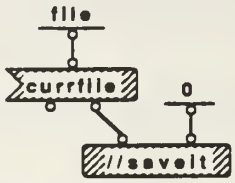
file/open... 1:1



After verifying the user's intentions
(if there have been any changes to
the original query) on what to
do with the current query, puts up the
get-file dialog to allow the user to
pick or enter a file to load, then
if it was a valid file load gdbobjlist
from it.

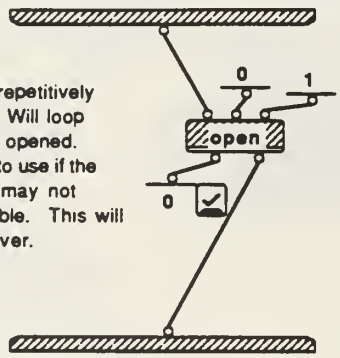
file/save 1:1

Save the data from gdbobjlist into the currfile.

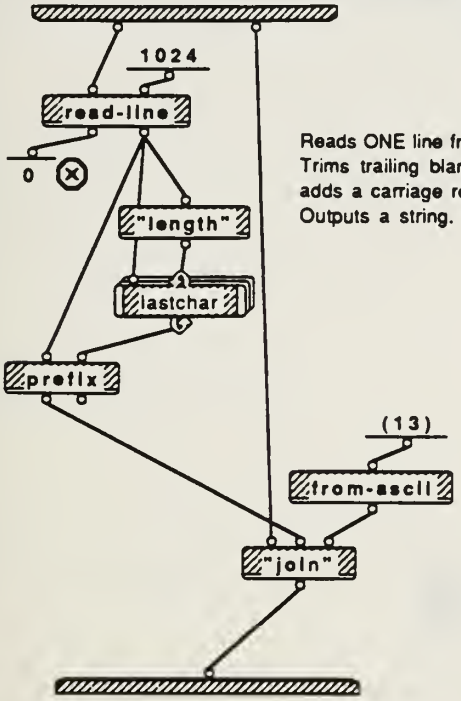


file/openloop 1:1

To be called repetitively to open a file. Will loop until the file is opened. DANGEROUS to use if the file requested may not become available. This will then loop forever.

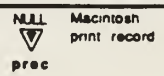


file/readtext 1:1



Reads ONE line from a file.
Trims trailing blanks and
adds a carriage return.
Outputs a string.

▽ dfqlprint



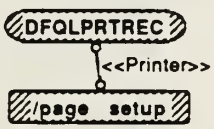
Macintosh
print record

dfqlprint

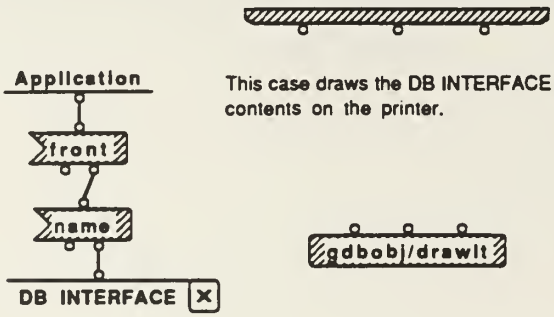


dfqlprint/prtsetup 1:1

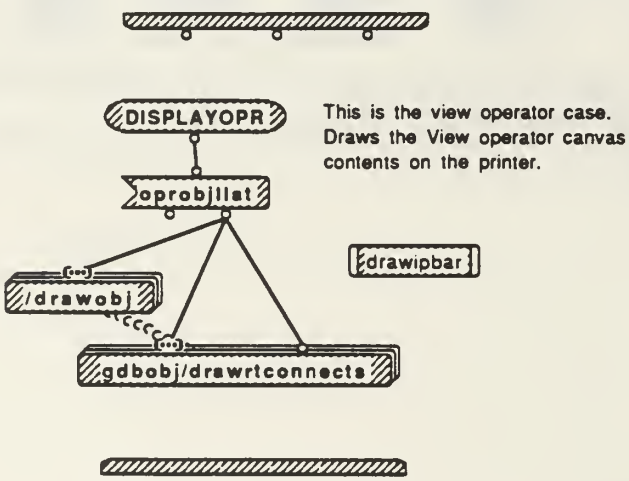
Change the setup for the printer record for DFQL printouts.



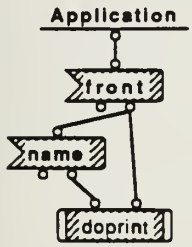
dfqlprint/draw page 1:2



dfqlprint/draw page 2:2

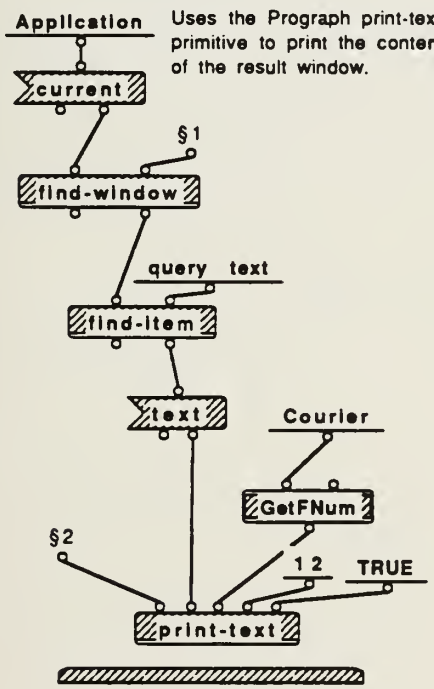


dfqlprint/pickprint 1:1



Based on what is currently the front window, call the appropriate print method.

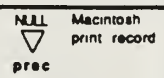
dfqlprint/resultprint 1:1



Uses the Prograph print-text primitive to print the contents of the result window.

§1. Query Results
§2. Query Results

▽ Printer



Printer

<<>> initialize printer instance
CALL THIS METHOD FROM INITIAL PROGRAM AND DEFINITELY BEFORE ANY PRINTING

dispose dispose of print structures
CALL THIS METHOD BEFORE ALLOWING A PRINTER INSTANCE TO BE GARBAGED

report check and report print errors

page size returns page size as a rectangle

count pages count number of pages
OVERSHADOW THIS METHOD TO DETERMINE NUMBER OF PAGES

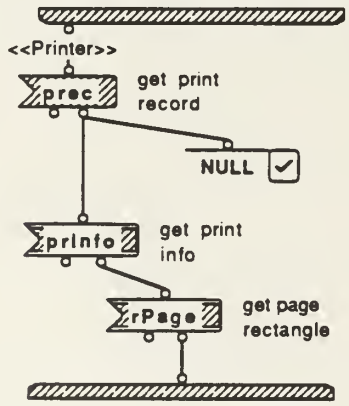
draw page draw contents of page
OVERSHADOW THIS METHOD TO DRAW YOUR PRINT PAGE

print begin the print process
CALL THIS METHOD WHEN YOU WANT TO PRINT, THAT IS IN RESPONSE TO THE Print... MENU ITEM

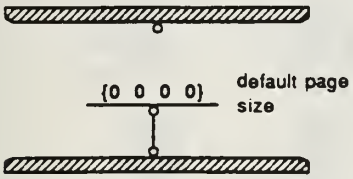
page setup bring up the page setup dialog
CALL THIS METHOD IN RESPONSE TO THE Page Setup... MENU ITEM

Printer class PROVIDED by TGSS.

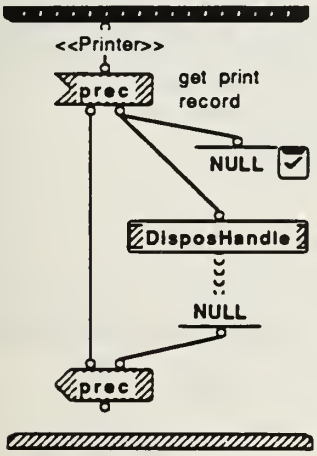
Printer/page size 1:2



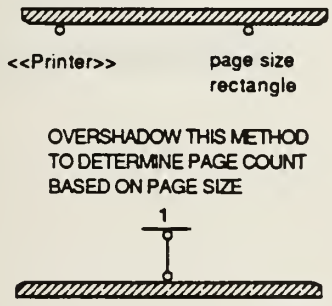
Printer/page size 2:2



Printer/dispose 1:1



Printer/count pages 1:1

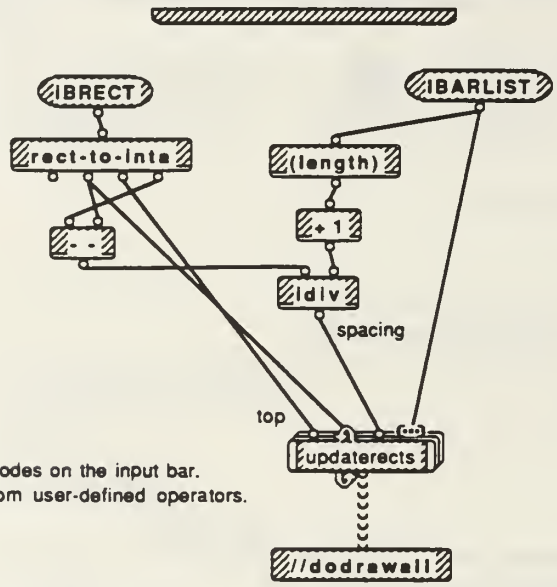


gdbobj

drawit mydraw myclick doerase dodraw centerrect drawrtconnecta drawirecta

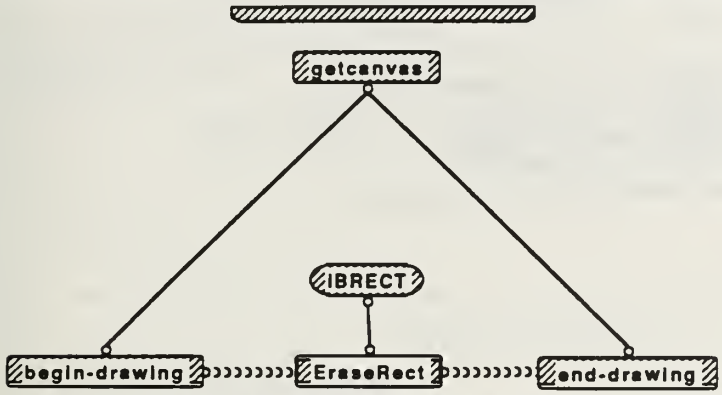
dodrawall drawallobj delete deselect drawinputbar eraseinputbar invert

gdbobj/drawirects 1:1



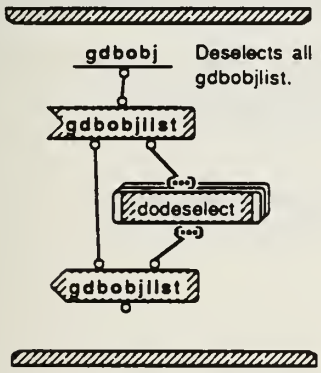
Draws nodes on the input bar.
Used from user-defined operators.

gdbobj/eraseinputbar 1:1



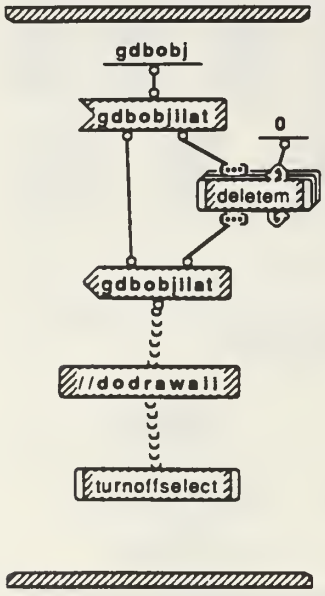
Erases the input bar from the canvas. Used at the termination of user defined ops screens.

gdbobj/deselect 1:1



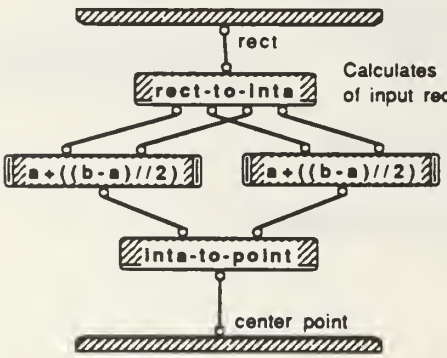
Deselects all selected objects in gdbobjlist.

gdbobj/delete 1:1



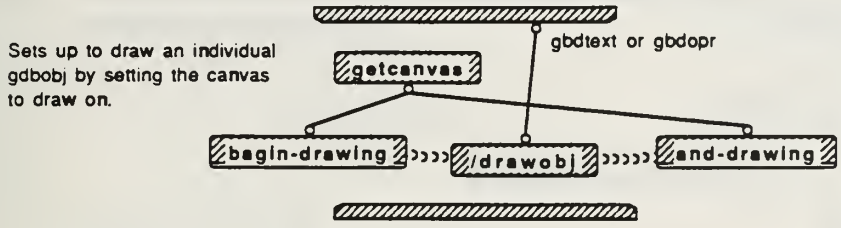
Executed in response to the Delete menu item. Goes through gdbobjlist and deletes all selected objects. After deletions redraws the canvas and then turns off the select option.

gdbobj/centerrect 1:1

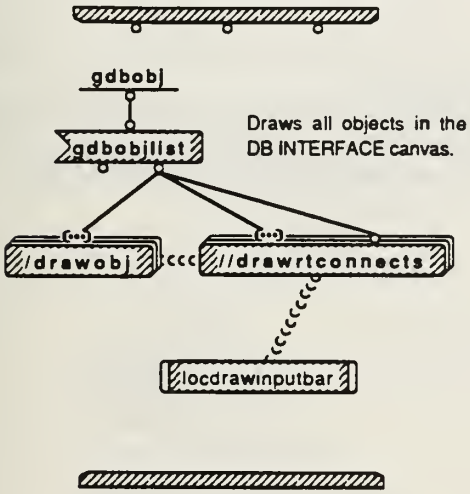


Calculates the center point (horiz) of input rectangle.

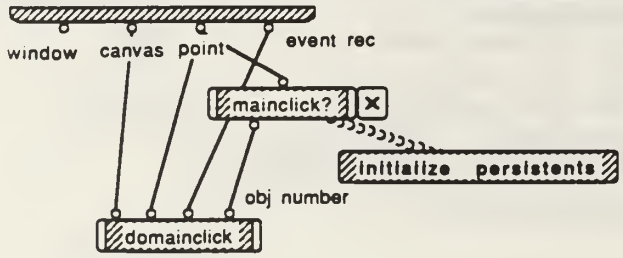
gdbobj/mydraw 1:1



gdbobj/drawit 1:1



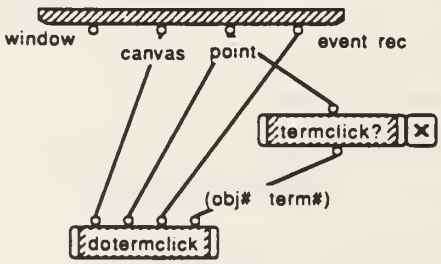
gdbobj/myclick 1:5



Handles clicks on the BODY of gdbobj objects.



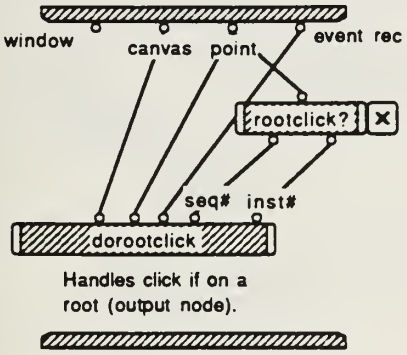
gdbobj/myclick 2:5



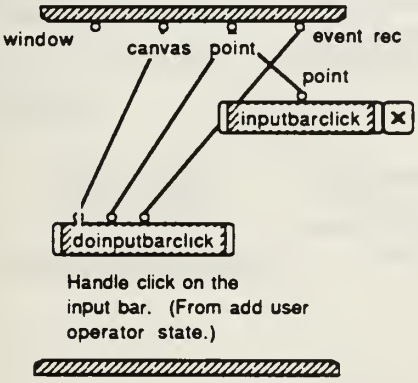
Click on a terminal (input node)



gdbobj/myclick 3:5



gdbobj/myclick 4:5



gdbobj/myclick 5:5

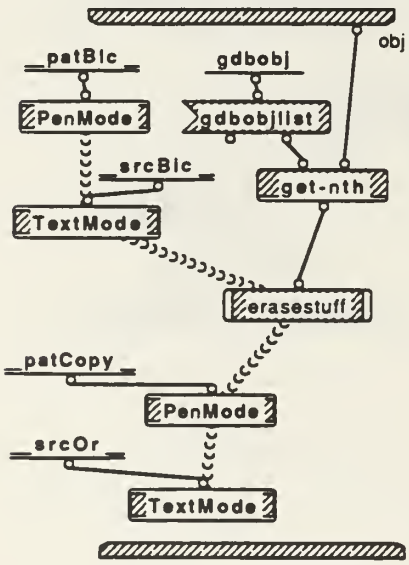


initialize persistents

If click anywhere in blank space reset the line drawing persistents. This will turn off lines that the user doesn't want to connect anymore.

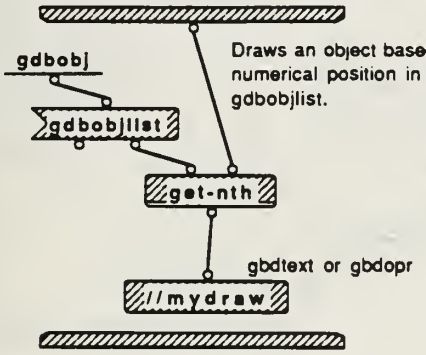


gdbobj/doerase 1:1

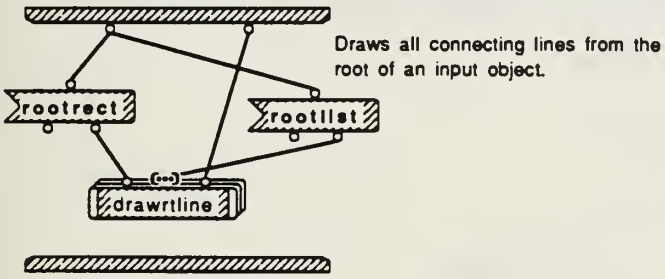


Erases the object based on the object number (the POSITION of the object in the list) NOT based on the instance number.

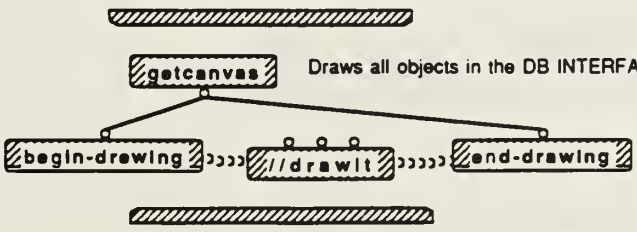
gdbobj/dodraw 1:1



gdbobj/drawrtconnects 1:1

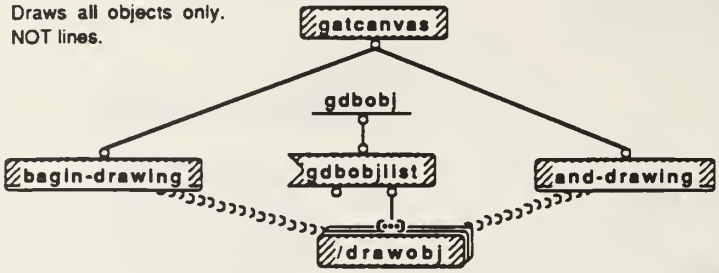


gdbobj/dodrawall 1:1

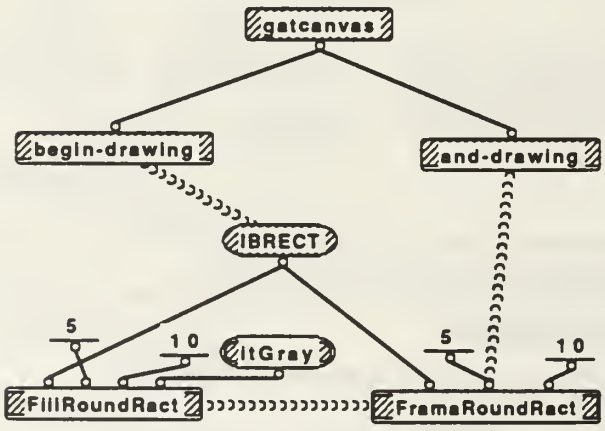


gdbobj/drawallobj 1:1

Draws all objects only.
NOT lines.

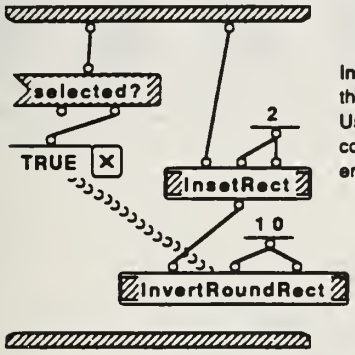


gdbobj/drawinputbar 1:1



Draws the input bar for the user-defined operators screens.

gdbobj/invert 1:2

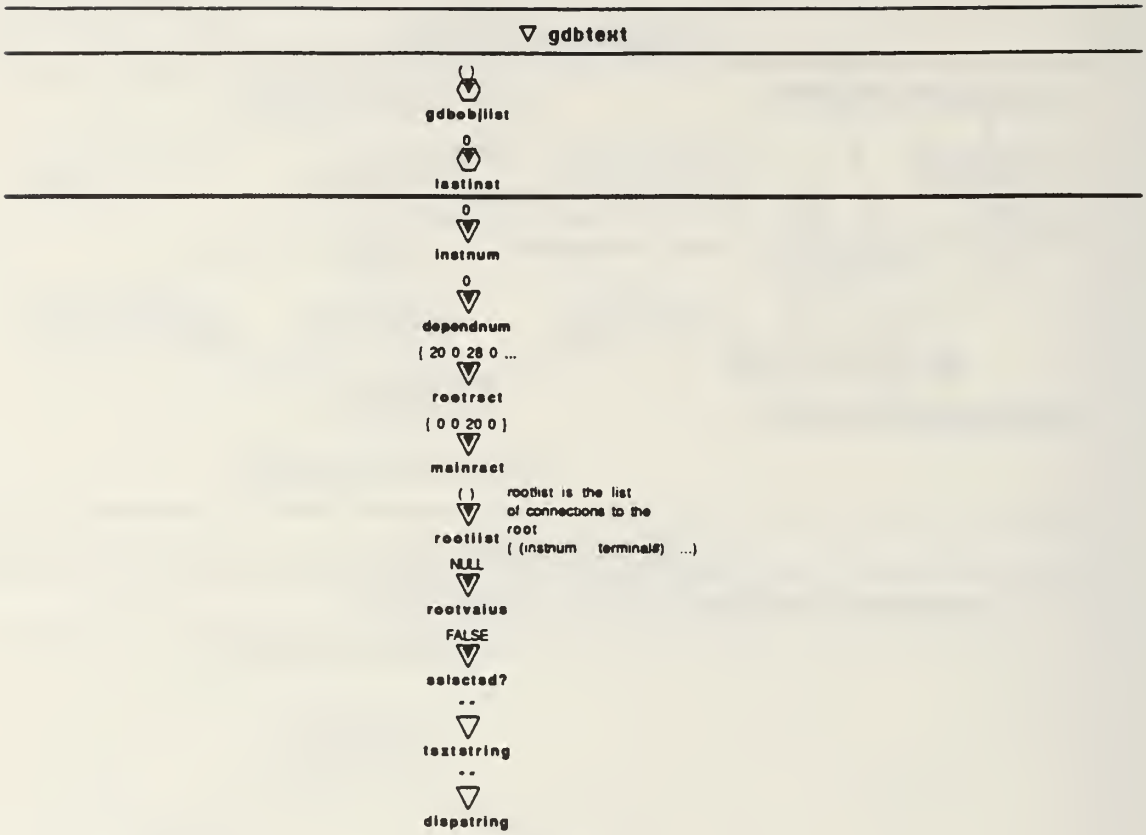


Inverts the color of an object if the object is selected.
Uses an inset first so that the corners of the object remain to enhance the appearance.

gdbobj/invert 2:2



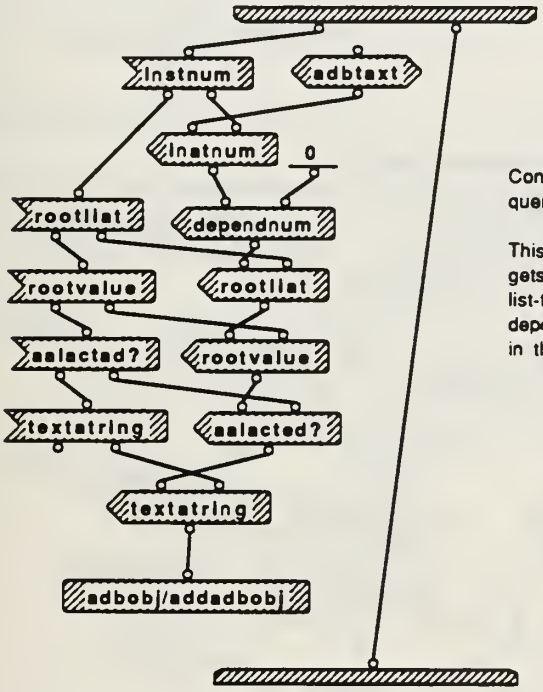



∇ gdbtext
gdbobjlist
0
lastinst
0
instnum
0
dependnum
{ 20 0 28 0 ...
rootract
{ 0 0 20 0 }
mainract
( )
rootlist
rootlist is the list of connections to the root
((instnum terminals) ...)
NULL
rootvalue
FALSE
selected?
..
textstring
..
dispstring

gdbtext



gdbtext/makeadbobj 1:1



Converts gdbtext to adbtext object for query graph.

This creation is all done with explicit gets and sets (rather than inst-to-list and list-to-inst) in order to ensure that it is not dependent on position of the attributes in the lists.

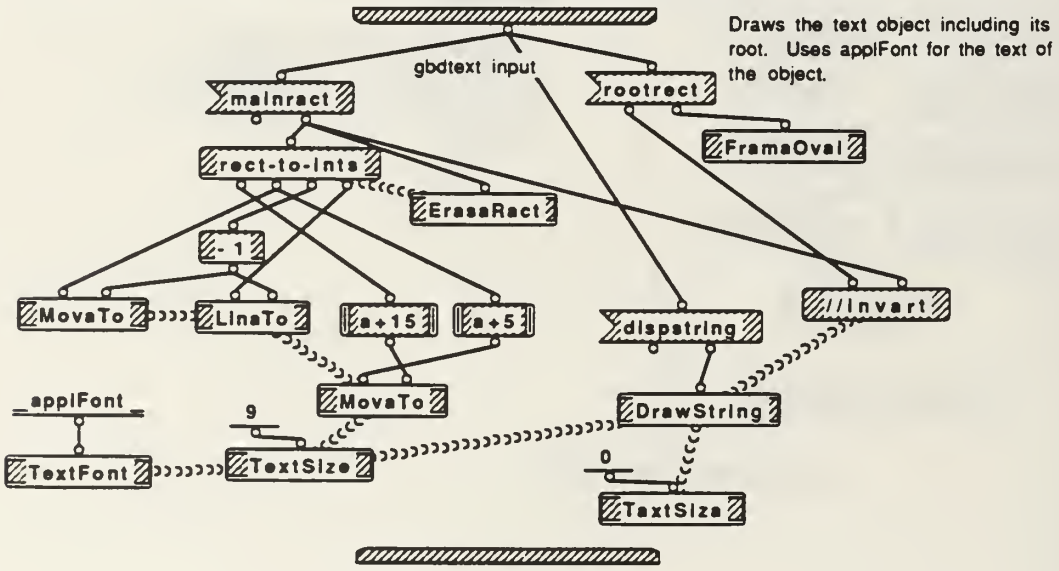
gdbtext/drawtconnects 1:1



pass in text obj
do nothing since
it has no terminals

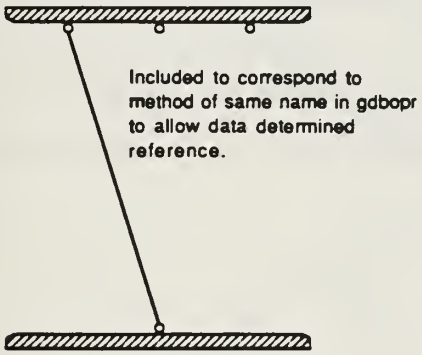


gdbtext/drawobj 1:1

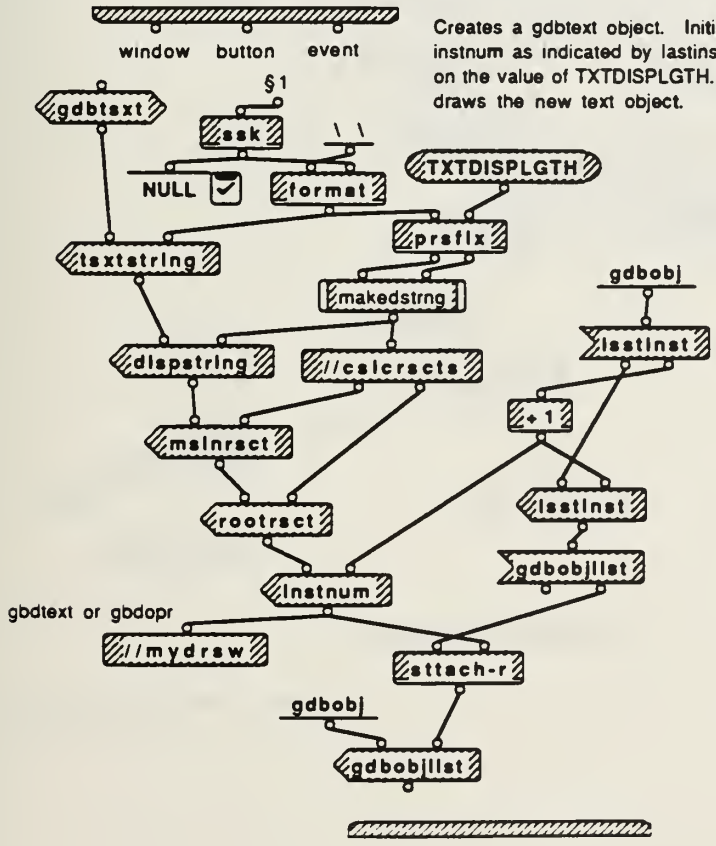


Draws the text object including its root. Uses applFont for the text of the object.

gdbtext/setterms 1:1



gdbtext/create 1:1

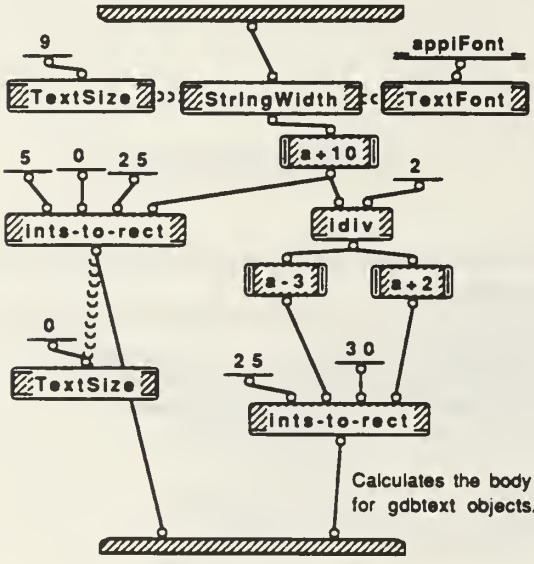


Creates a gdbtext object. Initiallizes instnum to the next sequential instnum as indicated by lastinst. Truncates text for display based on the value of TXTDISPLGTH. When done with initialization draws the new text object.

gdbtext/create 1:1

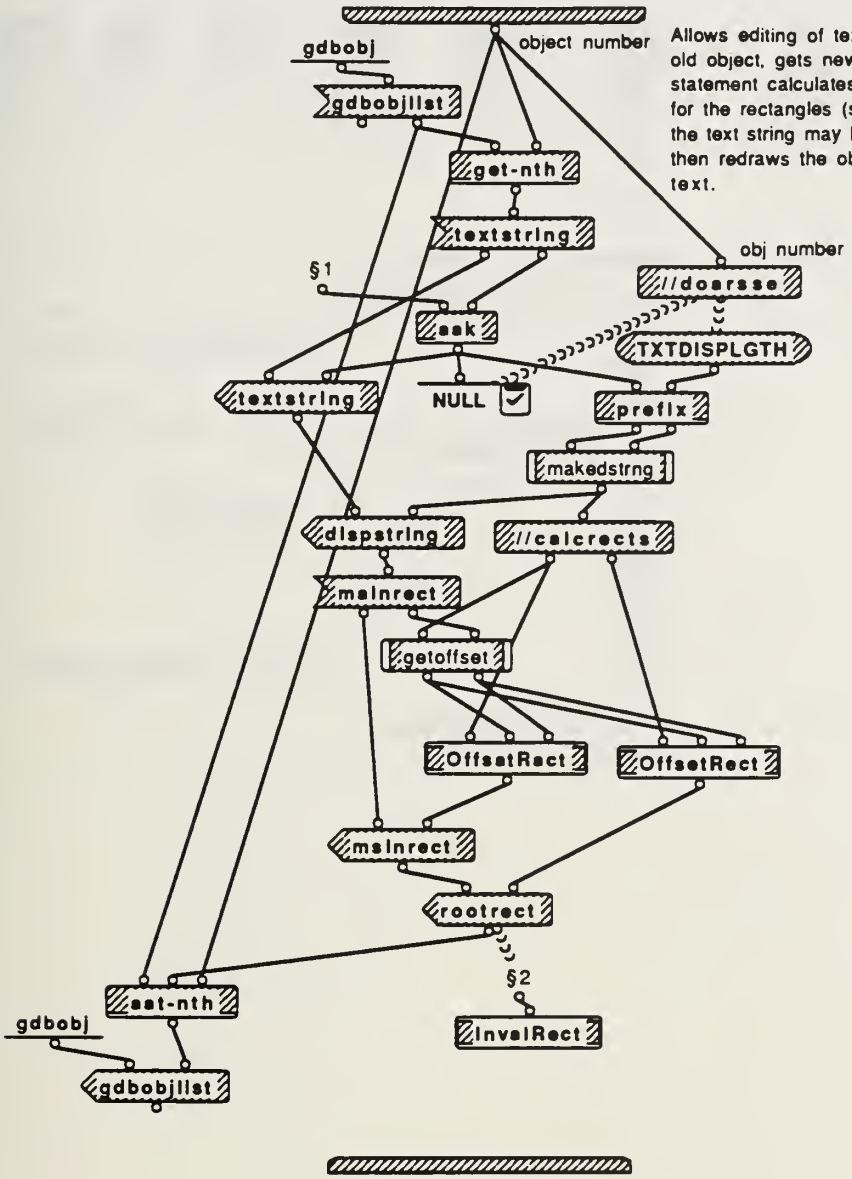
§1. Please enter your text.

gdbtext/calcrects 1:1



Calculates the body and root rectangle for gdbtext objects.

gdbtext/edittextobj 1:1



Allows editing of text object. Erases old object, gets new text from the ask statement calculates the new values for the rectangles (since the size of the text string may have changed) and then redraws the object with the new text.

§1. You may edit the string below.
§2. {0 0 3200 3200}

∇ gdbopr

( )
gdbobjlist

0
lastinst

0
instnum

0
dependnum

[20 0 28 0 ...
rootract

[0 0 20 0]
mainract

( )
rootlist

rootlist is the list of connections to the root
((instnum terminalid) ...)

..
rootvalue

FALSE
selected?

..
opname

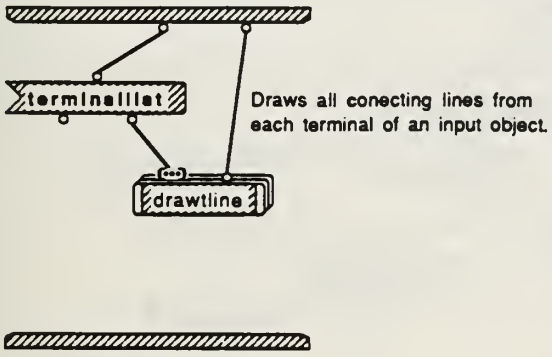
( )
terminallist

terminal connections to opr
((termrect from obj instnum) ...)

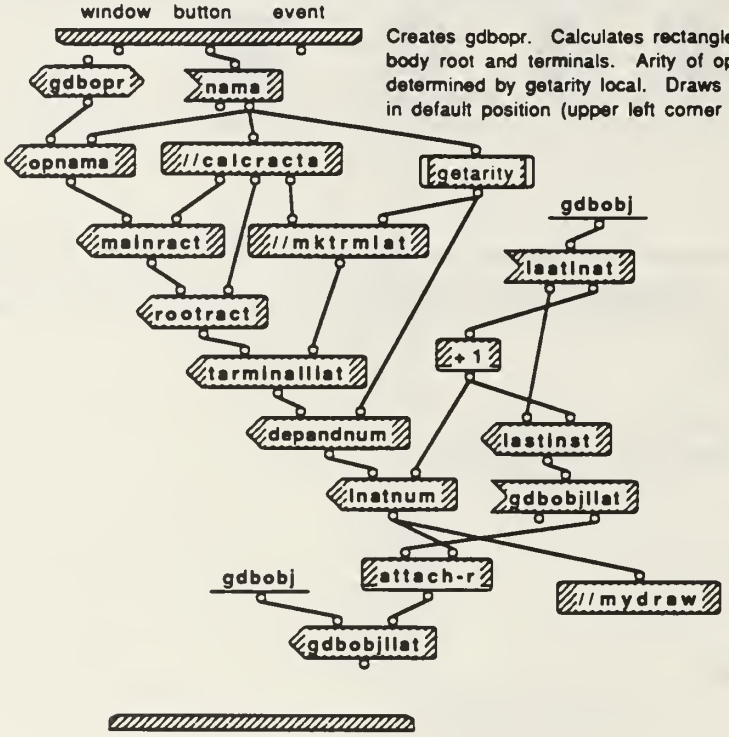
gdbopr



gdbopr/drawtconnects 1:1



gdbopr/create 1:1



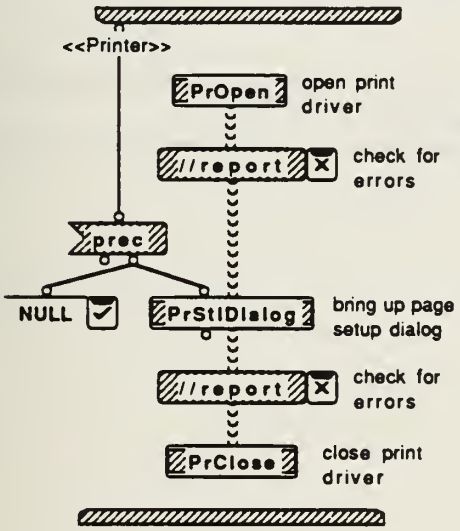
Creates gdbopr. Calculates rectangles for body root and terminals. Arity of operator determined by getarity local. Draws new op in default position (upper left corner of canvas).

Printer/draw page 1:1

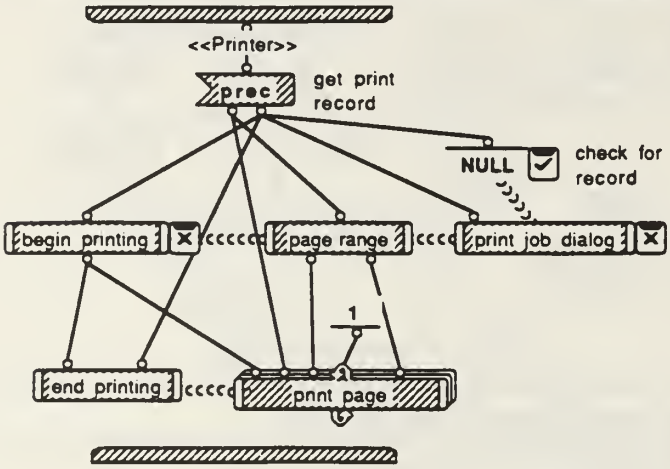
<<Printer>> page size rectangle page number

OVERSHADOW THIS METHOD
TO DRAW THE CONTENTS
OF YOUR PRINT PAGE

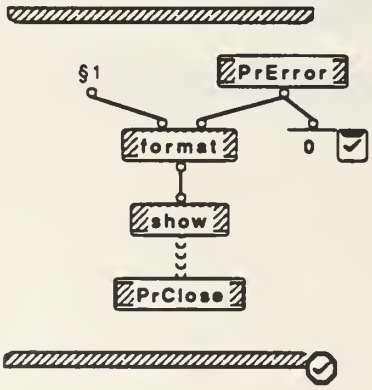
Printer/page setup 1:1



Printer/print 1:1

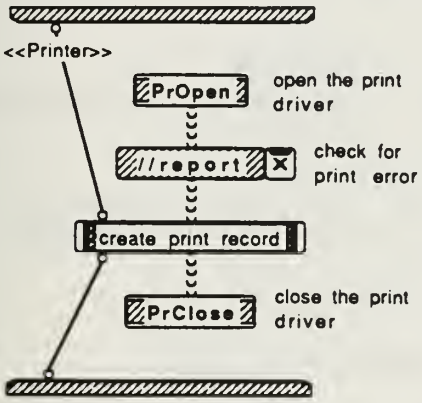


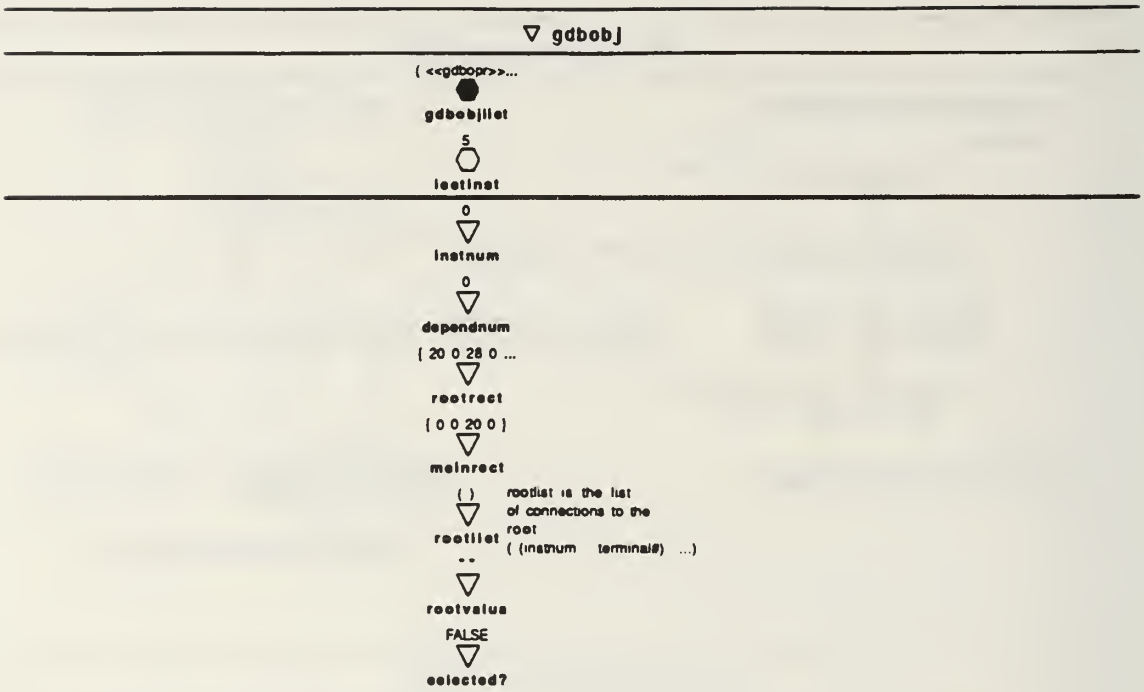
Printer/report 1:1



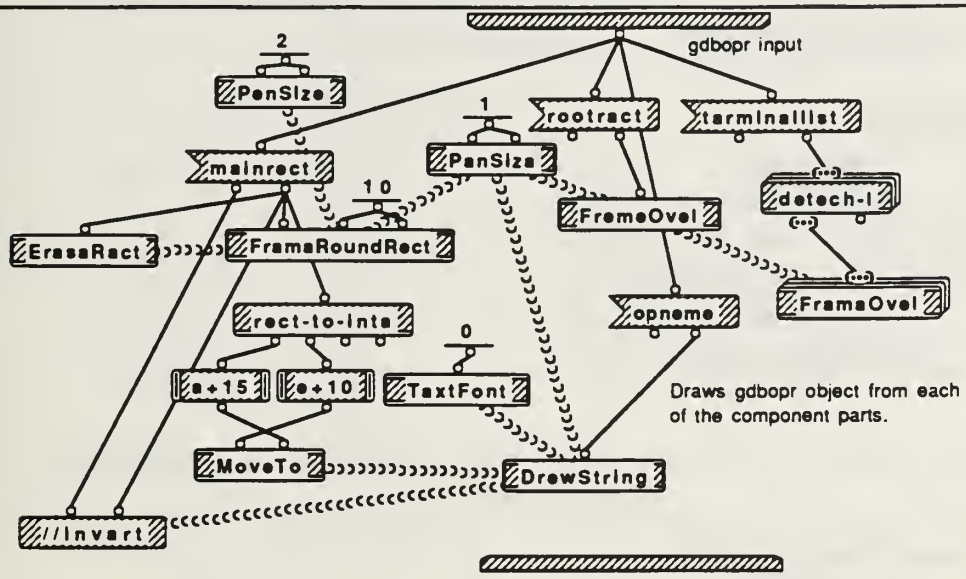
§1. Printer Error #\\

Printer/<<>> 1:1

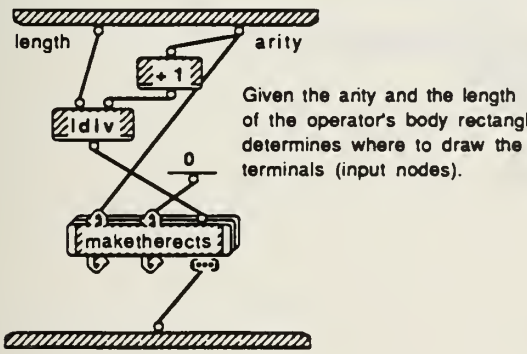

<<Printer>>
PrOpen
open the print driver
//report
check for print error
create print record
PrClose
close the print driver


▽ gdbobj
(<<gdboopr>>...
gdbobjlist
5
leetinst
0
instnum
0
dependnum
(20 0 28 0 ...
rootrect
(0 0 20 0)
mainrect
( )
rootlist
rootlist is the list of connections to the root
((instnum terminal#) ...)
rootvalue
FALSE
selected?

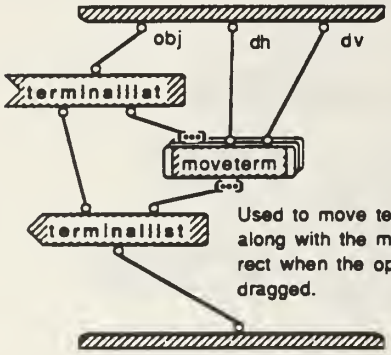
gdbopr/drawobj 1:1



gdbopr/mktrmlst 1:1

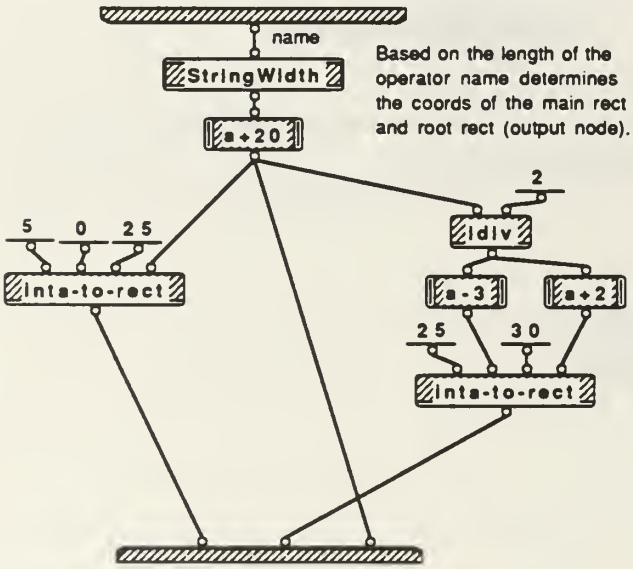


gdbopr/setterms 1:1



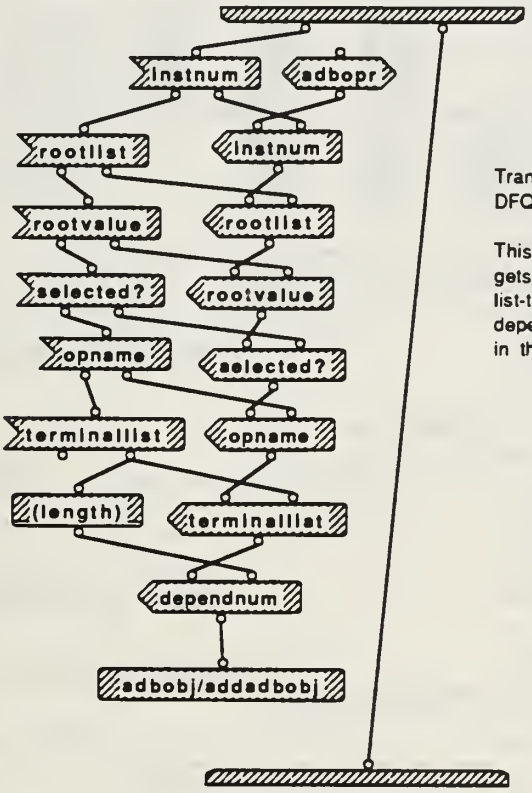
Used to move terminals along with the main (body) rect when the opr is dragged.

gdbopr/calcrects 1:1



Based on the length of the operator name determines the coords of the main rect and root rect (output node).

gdbopr/makeadbobj 1:1



Transforms a gdbopr into an adbopr for the DFQL query graph.

This creation is all done with explicit gets and sets (rather than inst-to-list and list-to-inst) in order to ensure that it is not dependent on position of the attributes in the lists.

∇ usropr

( )
gdbobjlist

0
lastinst

0
instnum

0
dependnum

[20 0 28 0 ...
rootract

[0 0 20 0]
mainract

( ) rootlist is the list of connections to the root
rootlist ((instnum terminal#) ...)

- -
rootvalue

FALSE
selected?

- -
opname

( ) terminal connections to opr
terminallist ((termract from obj instnum) ...)

( ) dummy inst nums and connection(s) to internal insts
termconlist ((dum# ((inst# term#)(inst# term#))) (dum#((in# t#))))

0 internal inst num that is connected to the root
rootcon

( ) gdbobjlist from screen to allow display
oprobjlist

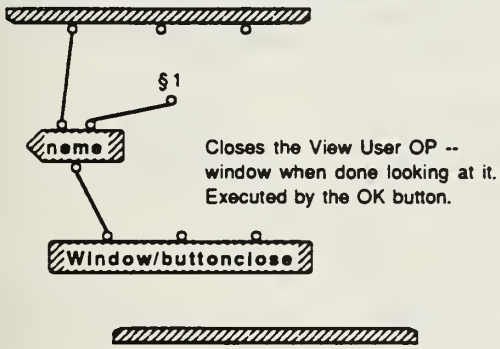
0 last instance number used in this user opr
oprlastinst

- - helptext is entered by the user when the user operator is defined
helptext

usropr

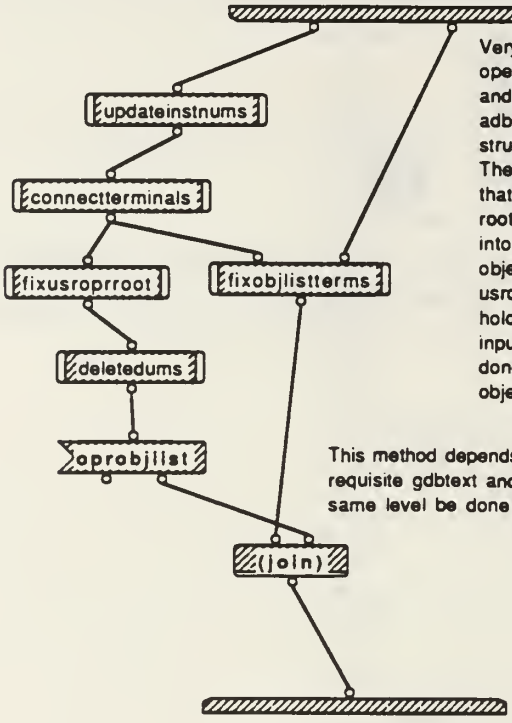


usropr/viewclose 1:1



§1. View User Operator

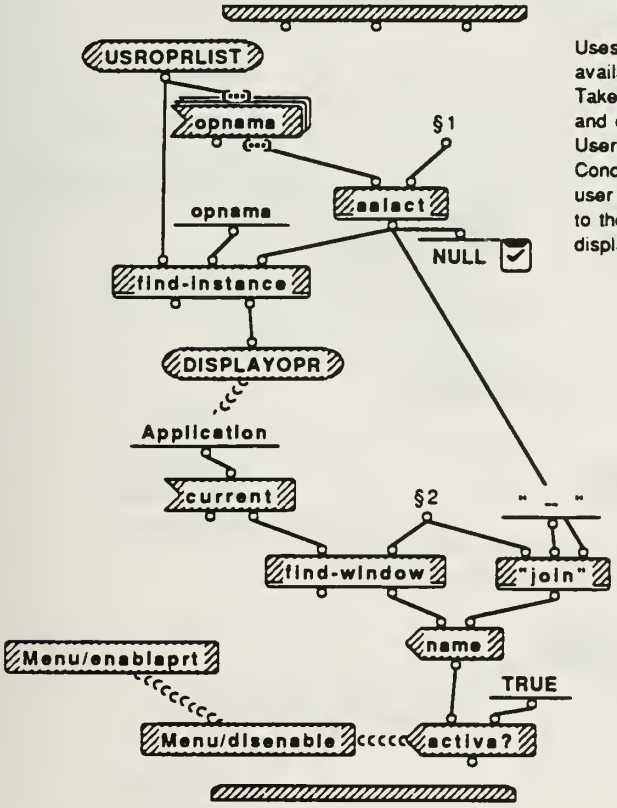
usropr/makeadbobj 1:1



Very complicated method which takes a user defined operator (possibly containing other user defined ops) and turns it into its constituent text objects and primitive adbobj's. First updates the instnums of the embedded structure of the user op (the user op's internal gdbobjlist). Then connects the terminals from the user op to the objects that they should go to internally. Also connect the internal root to the appropriate external objects. Then go back into the external list and fix redirect all of the external objects connections to the instnums of the internal usrobj object instances. Finally delete the dummy place holder variables (that provide the actual nodes on the input bar -- really each node is a dummy text object -- done for expediency) and join all of the newly created objects into the old incoming list and output it.

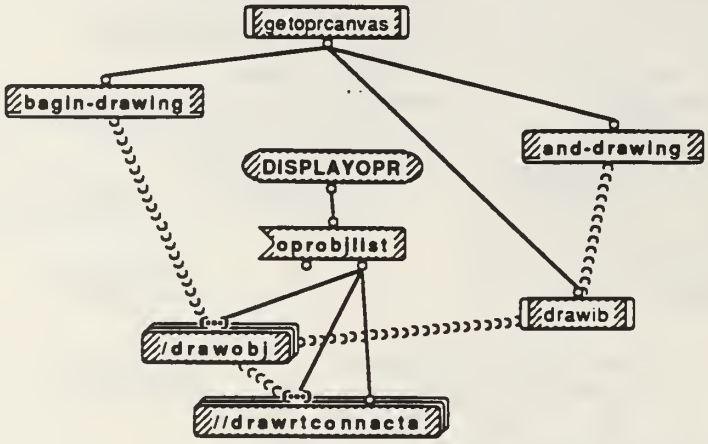
This method depends on all of the requisite gdbtext and gdboprs at the same level be done first.

usropr/viewop 1:1



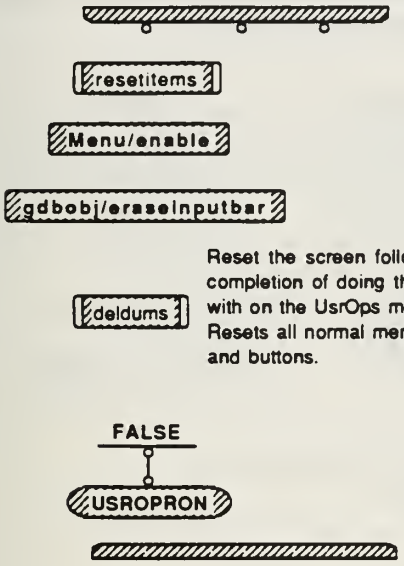
Uses select dialog to display available user operators. Takes the one that was selected and displays it in the View User Operator -- window. Concatenates the name of the user operator being displayed to the window name so it gets displayed in the title bar.

§1. CHOOSE OPERATOR
§2. View User Operator



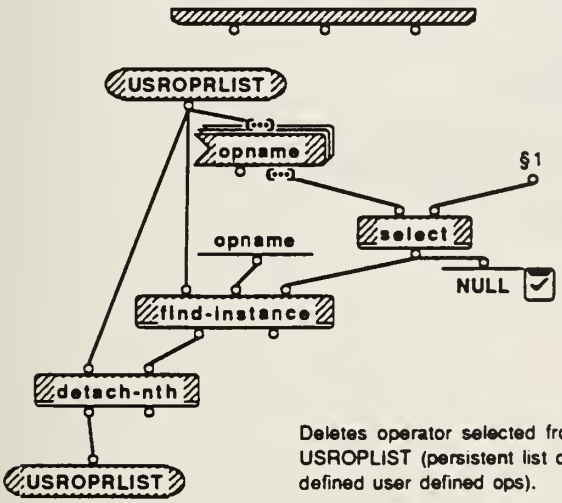
Basically the same as gdbopr draw except it must account for the input bar and any connections to it.

usropr/cancel 1:1



Reset the screen following completion of doing things with on the UsrOps menu. Resets all normal menu choices and buttons.

usropr/delop 1:1

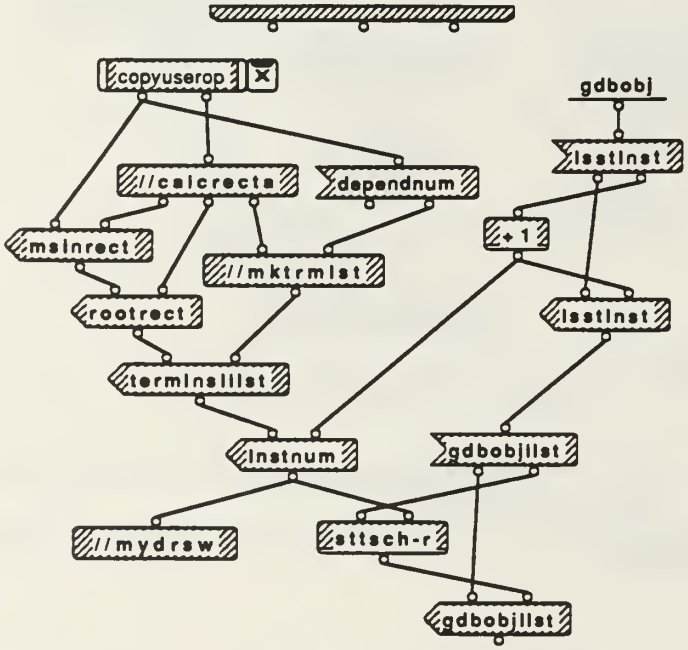


Deletes operator selected from the USROPLIST (persistent list of all the defined user defined ops).

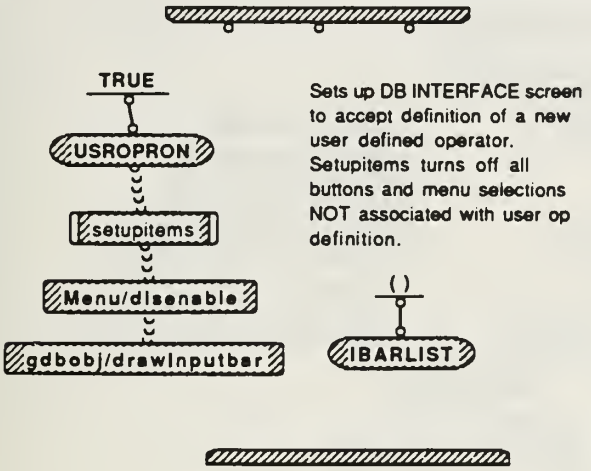
usropr/delop 1:1

§1. DELETE OPERATOR

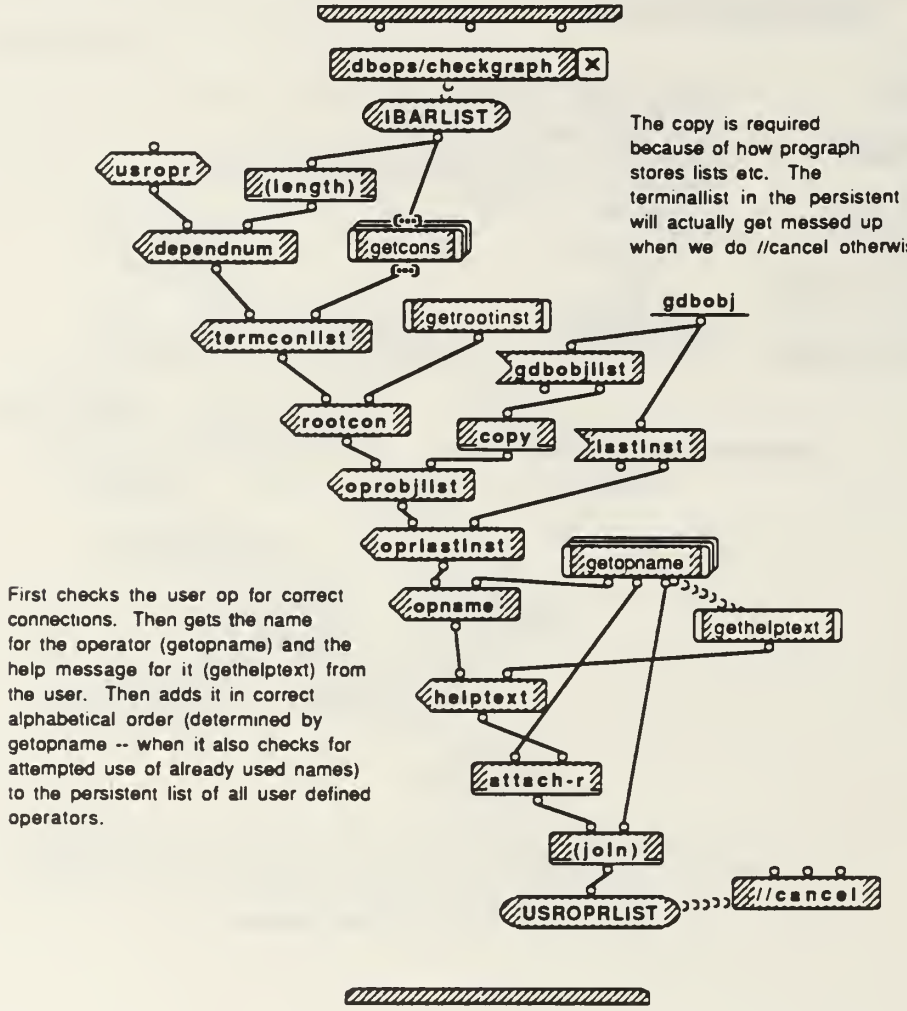
usropr/selop 1:1



Called in response to the select a usropr menu item. Gives the user a list of currently defined usr ops. Take the one that he picked (comes out of copyuserop) and create it visually (calculate its rects) and add it to the gdbobjlist.

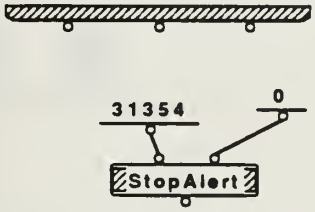

TRUE
USROPRON
setupitems
Menu/disenable
gdbobj/drawInputbar
Sets up DB INTERFACE screen to accept definition of a new user defined operator. Setupitems turns off all buttons and menu selections NOT associated with user op definition.
( )
IBARLIST

usropr/storeop 1:2



The copy is required because of how prograph stores lists etc. The terminallist in the persistent will actually get messed up when we do //cancel otherwise.

First checks the user op for correct connections. Then gets the name for the operator (getopname) and the help message for it (gethelptext) from the user. Then adds it in correct alphabetical order (determined by getopname -- when it also checks for attempted use of already used names) to the persistent list of all user defined operators.



Alert should say that terminals and roots are not connected correctly. User should check his query graph.

∇ gdbdsp

( )

gdbobjlist

0

lastinst

0

instnum

0

dependnum

{ 20 0 28 0 ...

rootrect

{ 0 0 20 0 }

mainrect

( )

rootlist

rootlist is the list of connections to the root
((instnum terminals) ...)

• •

rootvalue

FALSE

selected?

• •

opname

( )

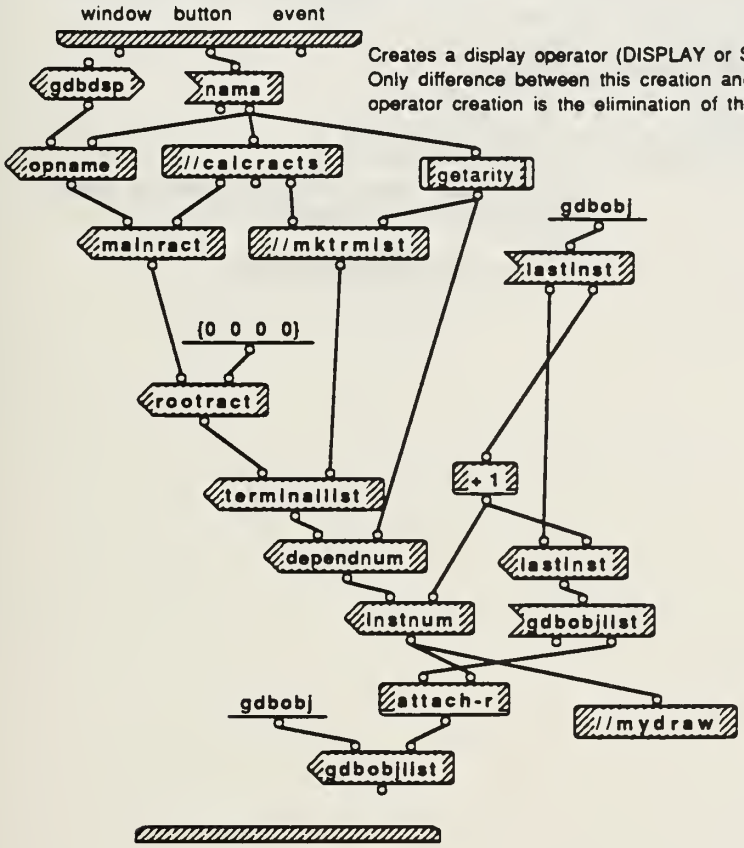
terminallist

terminal connections to opr
((termrect from obj instnum) ...)

gdbdsp

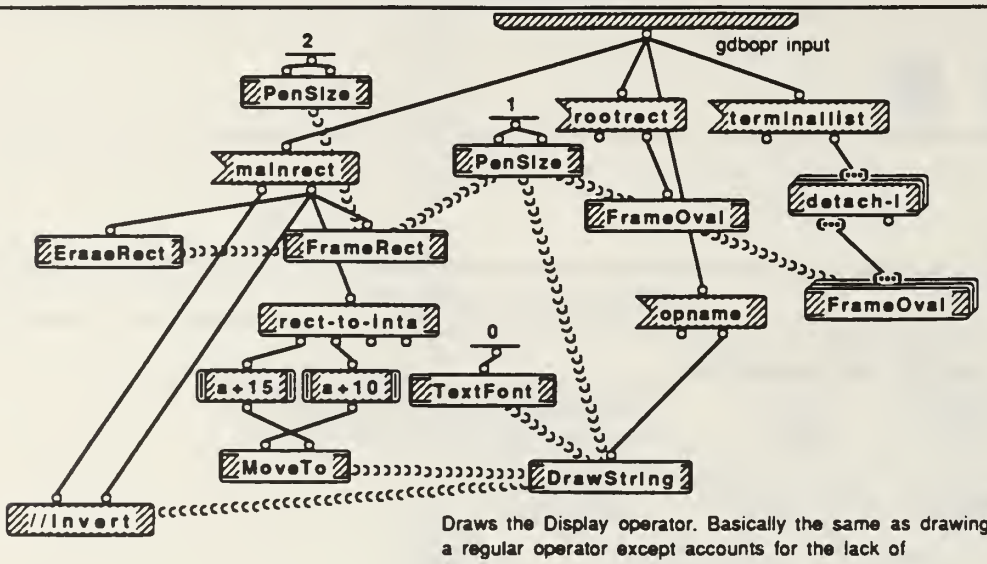


gdbdsp/create 1:1



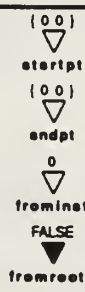
Creates a display operator (DISPLAY or SDISPLAY). Only difference between this creation and regular operator creation is the elimination of the root.

gdbdsp/drawobj 1:1



Draws the Display operator. Basically the same as drawing a regular operator except accounts for the lack of a root.

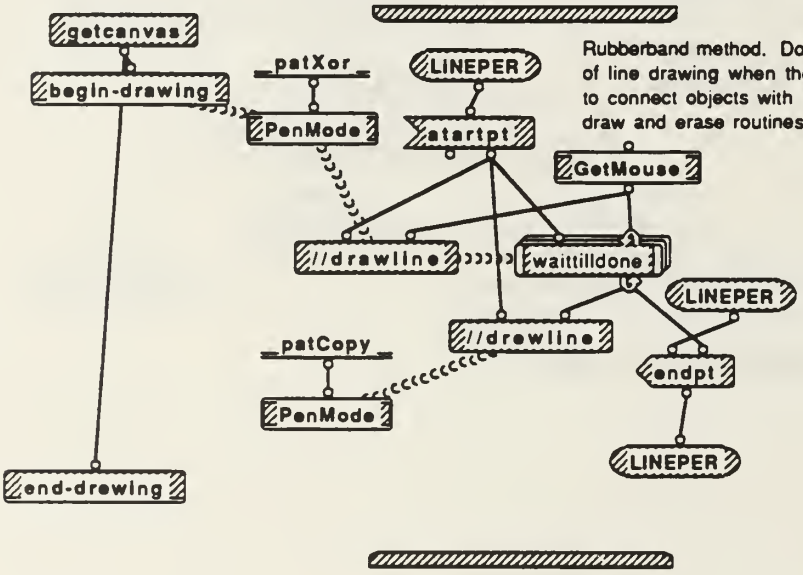
▽ lineobj



lineobj

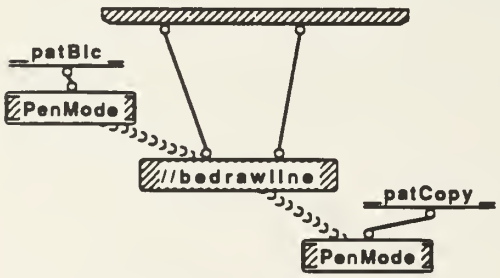


lineobj/rubberb 1:1



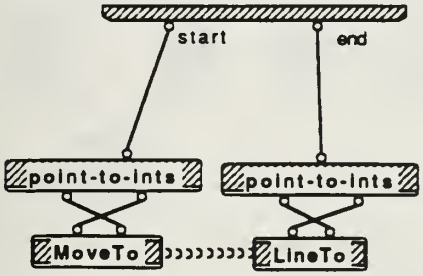
Rubberband method. Does the rubberband of line drawing when the user is trying to connect objects with lines. Uses continuous draw and erase routines. Until a mouse click.

lineobj/eraseline 1:1



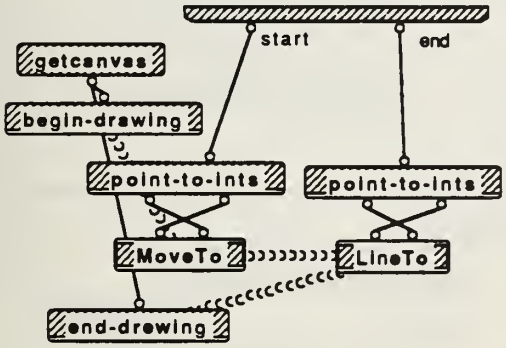
Erases a line from point1 to point2 by changing the pen mode. Changes pen back to patCopy when done.

lineobj/drawline 1:1

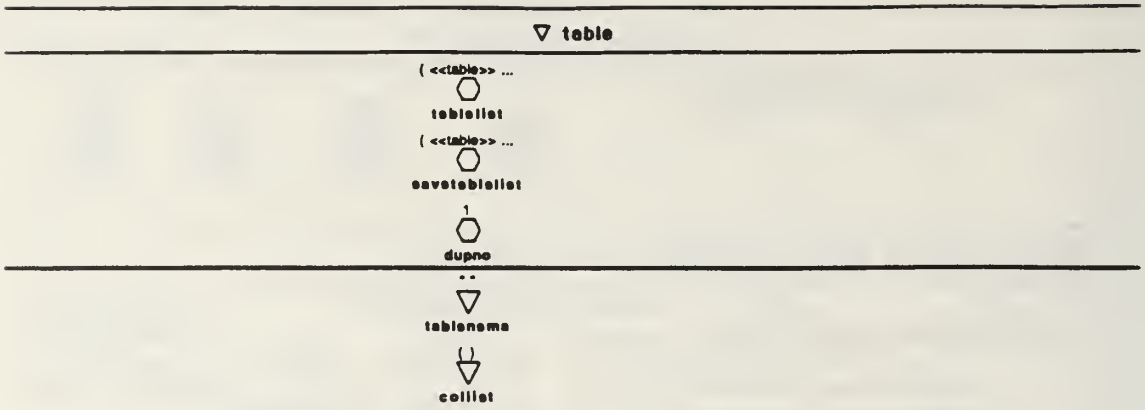


Draws a line from start point to end point.

lineobj/bedrawline 1:1



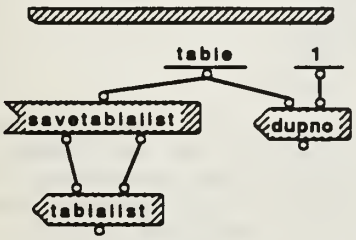
Same as drawline but includes the begin and end drawing primitives.


∇ table
(<<table>> ...
tablelist
(<<table>> ...
savetablelist
1
dupno
..
∇
tablename
( )
∇
collist

table

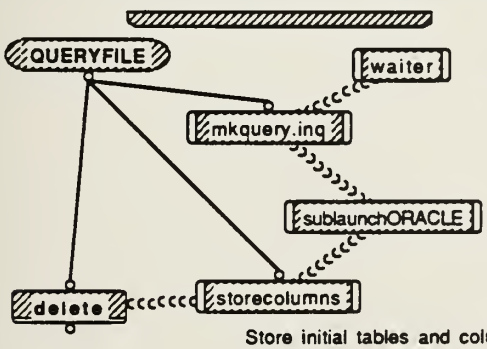


table/resetlist 1:1



Resets tablelist to only the initial relations found in savetablelist. These are the "REAL" relations in the database.

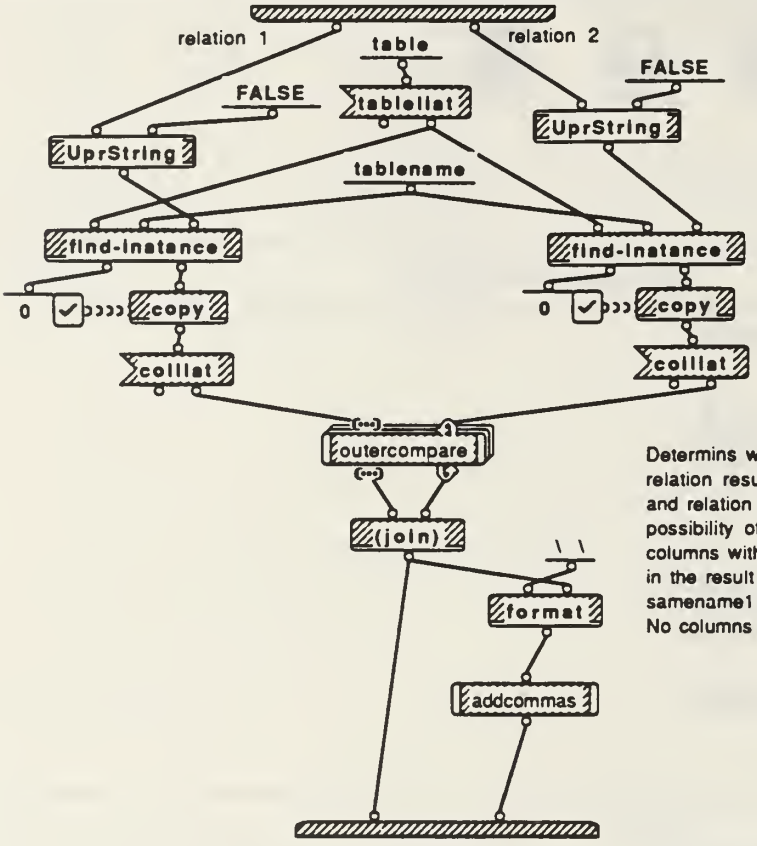
table/loadtable 1:1



Performs initial load of column names for all defined tables by going out to ORACLE and querying the columns relation for all the tables and columns that the user has access to.

Store initial tables and columns in the savetablelist where they will not be messed with for the duration of the program.

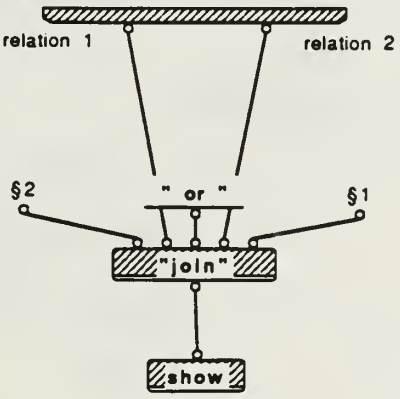
table/joincols 1:2



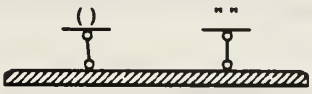
Determins what columns will appear in the relation resulting from a join of relation 1 and relation 2. Takes into account the possibility of the two input relaitons having columns with the same column names -- in the result you will get samename and samename1 (with "1" appended) columns. No columns are discarded in a join.

table/joincols 2:2

Could make this an alert.

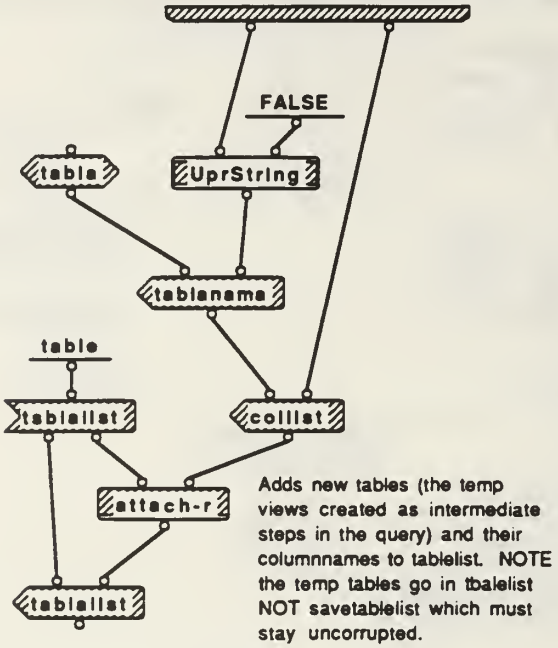


This error message shows up when a tablename is requested that doesn't exist. Execution will continue -- This won't stop the query from going to ORACLE -- if wanted you could stop this by using some fail notations from here up.



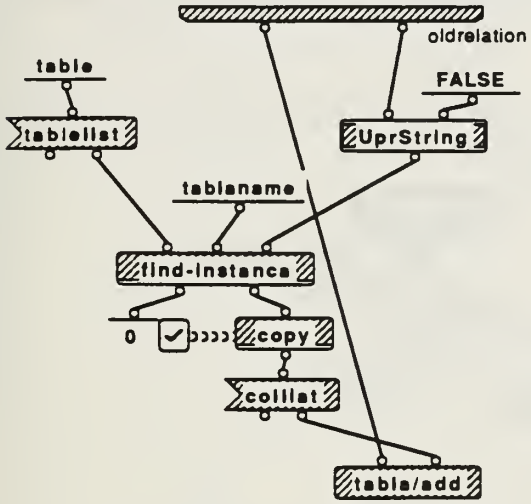
§1. " is an invalid table or view."
§2. "ERROR! Either "

table/add 1:1



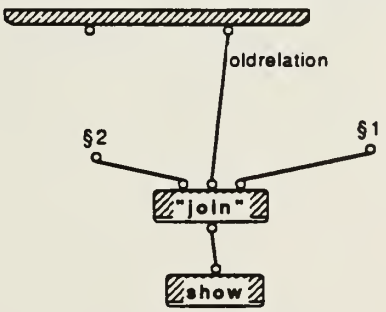
Adds new tables (the temp views created as intermediate steps in the query) and their columnnames to tablelist. NOTE the temp tables go in tbalelist NOT savetablelist which must stay uncorrupted.

table/addsame 1:2



Shorthand routine to add ALL of the columns from oldrelation to another incoming relation.

table/addsame 2:2

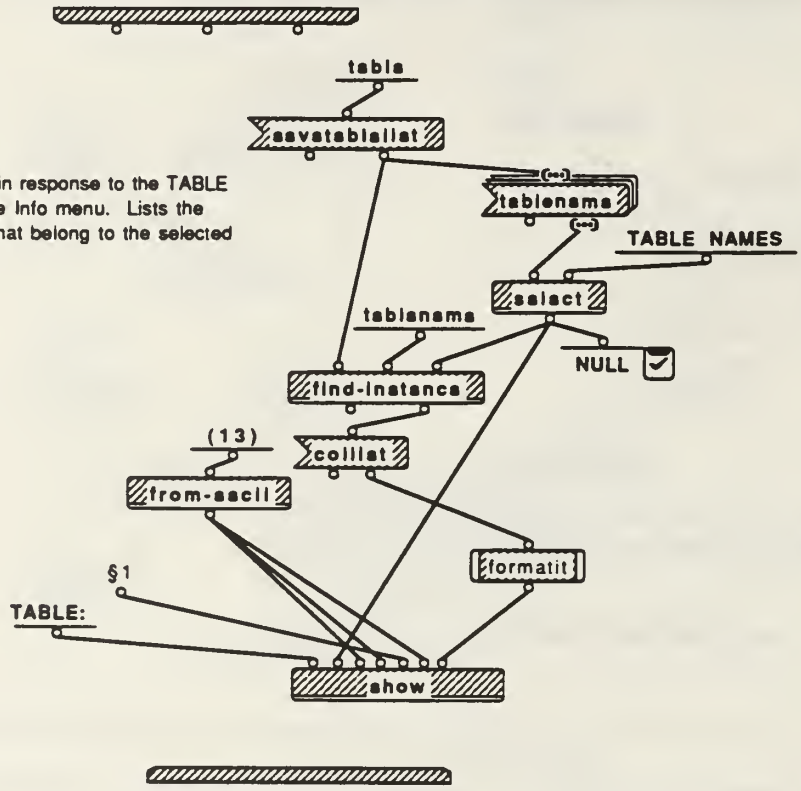


Could not find old relation name.

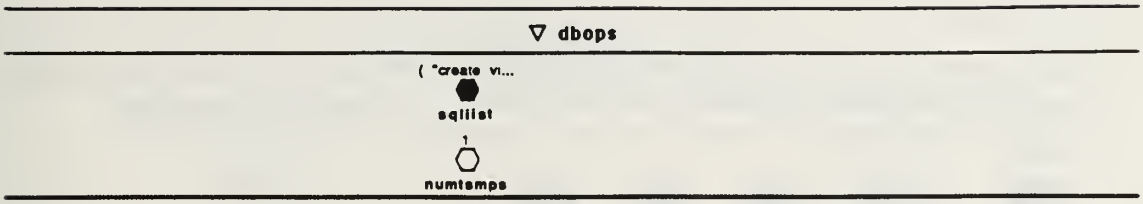
§1. " is invalid."
§2. "ERROR! Table or view "

table/colinfo 1:1

Executed in response to the TABLE item of the Info menu. Lists the columns that belong to the selected table.



§1. COLUMN NAMES:


∇ dbops
("create vi...
sqllist
1
numtemps

These are the main ENGINE type methods for the operation of the DFQL interpreter. makequery is the controlling method for execution of queries.



The methods with names the same as DFQL primitives convert those primitives into SQL code. The inputs to these methods are the SAME in the same order as that to the DFQL primitives. All of these methods (except the group satisfy methods which are documented more inside the method) simply concatenate the inputs to produce valid SQL statements.



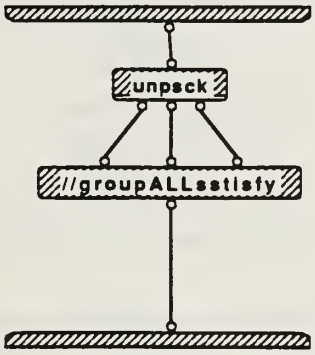


The "l" methods simply take input that consists of a DFQL operator name and a list of input arguments and calls the appropriate method after converting the input list into the inputs required for that method. This is done so that the actual primitive methods above are orthogonal to the DFQL operators that represent them. (The list unpacking COULD be done in the above methods -- but is not recommended because the above methods can also be used by the PROGRAPH programmer to make new primitives [see groupALLsatisfy and groupNsatisfy above], it is much better to take meaningful inputs than just a list.)

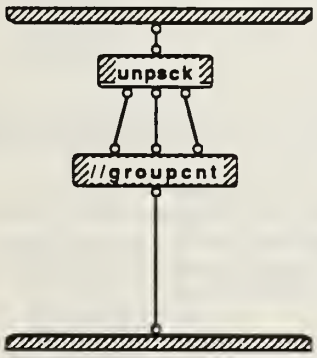




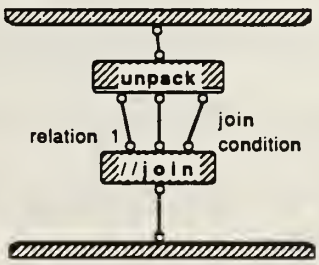
dbops/lgroupALLsatisfy 1:1



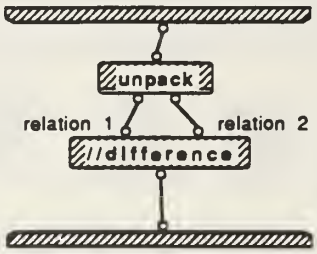
dbops/lgroupcnt 1:1



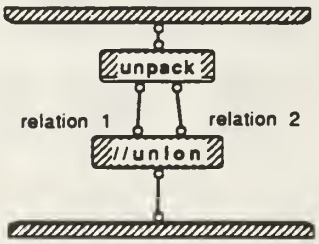
dbops/ljoin 1:1



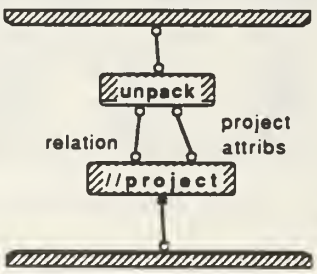
dbops/ldiff 1:1



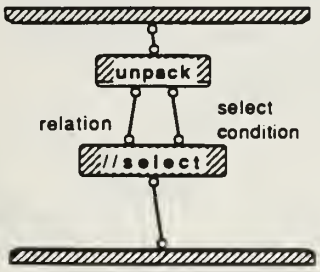
dbops/lunion 1:1



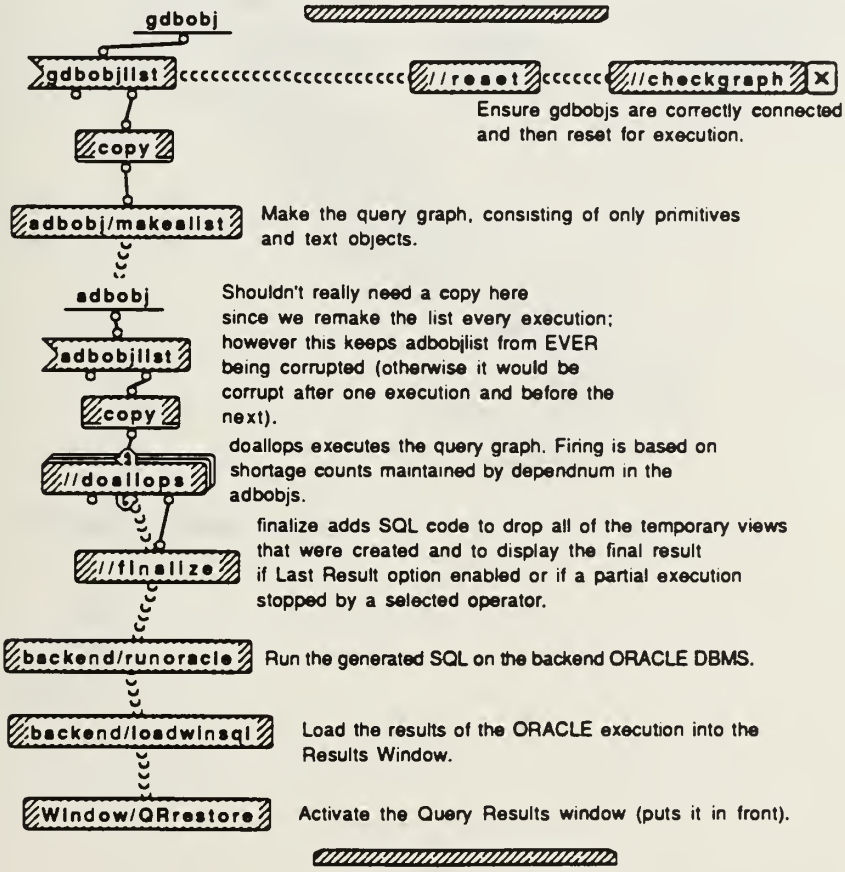
dbops/lproject 1:1



dbops/lselect 1:1



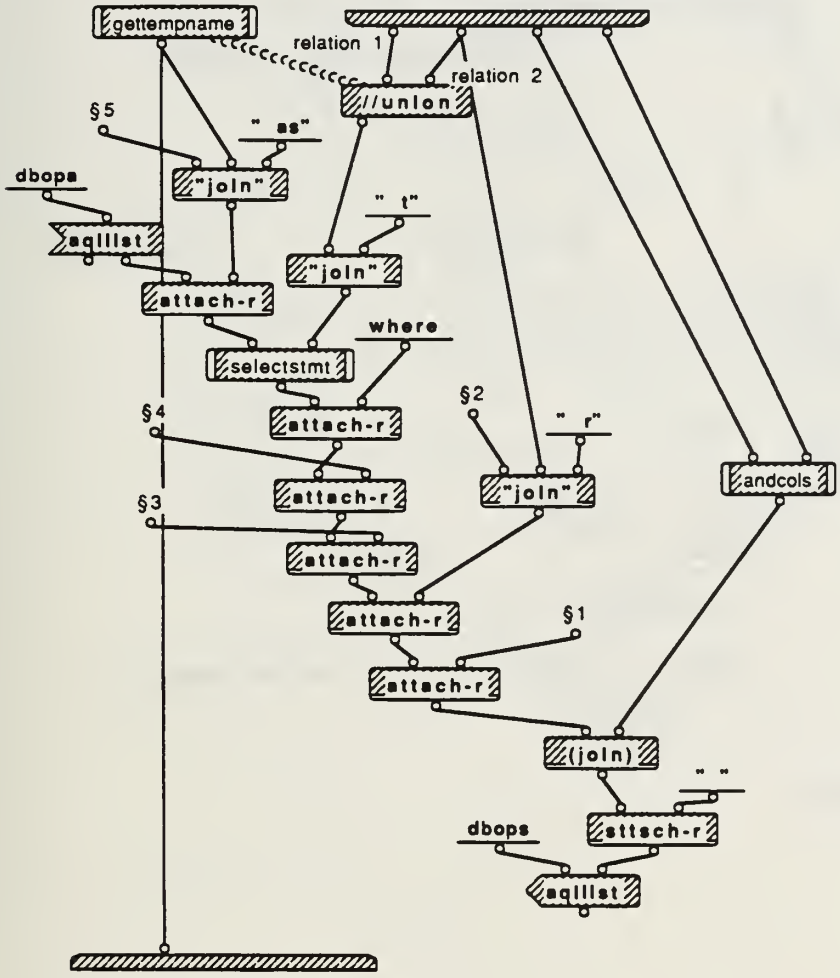
dbops/makequery 1:2





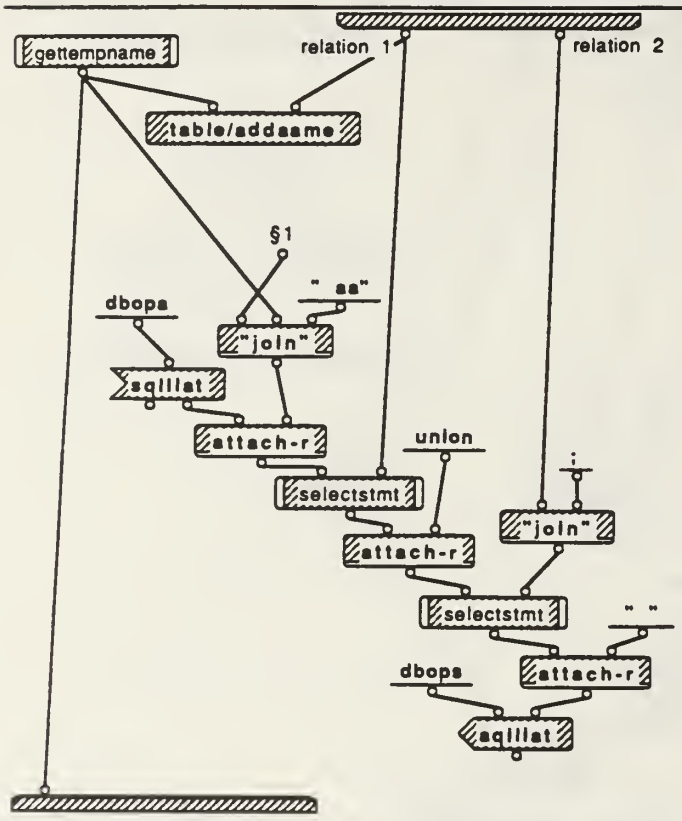
Alert should say that terminals and roots are not connected correctly. User should check his query graph.

dbops/gdifference 1:1



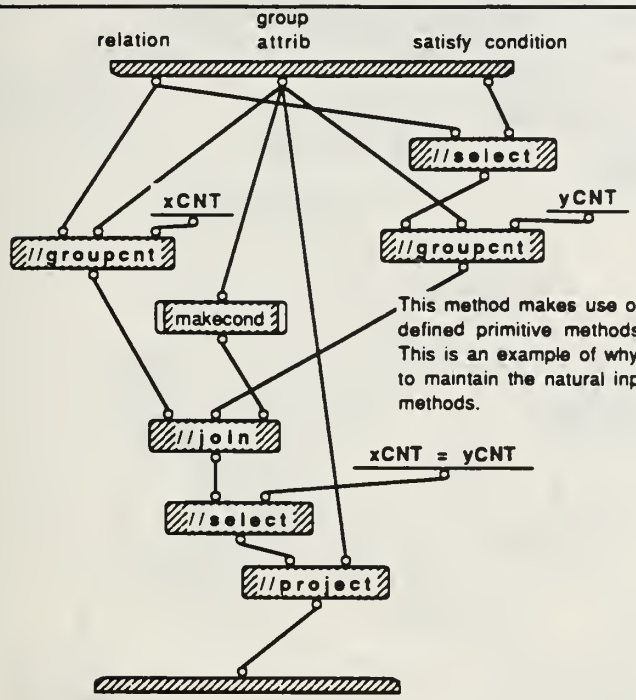
§1. " where"
§2. " from "
§3. " (select ""
§4. " not exists"
§5. "create view "

dbops/union 1:1

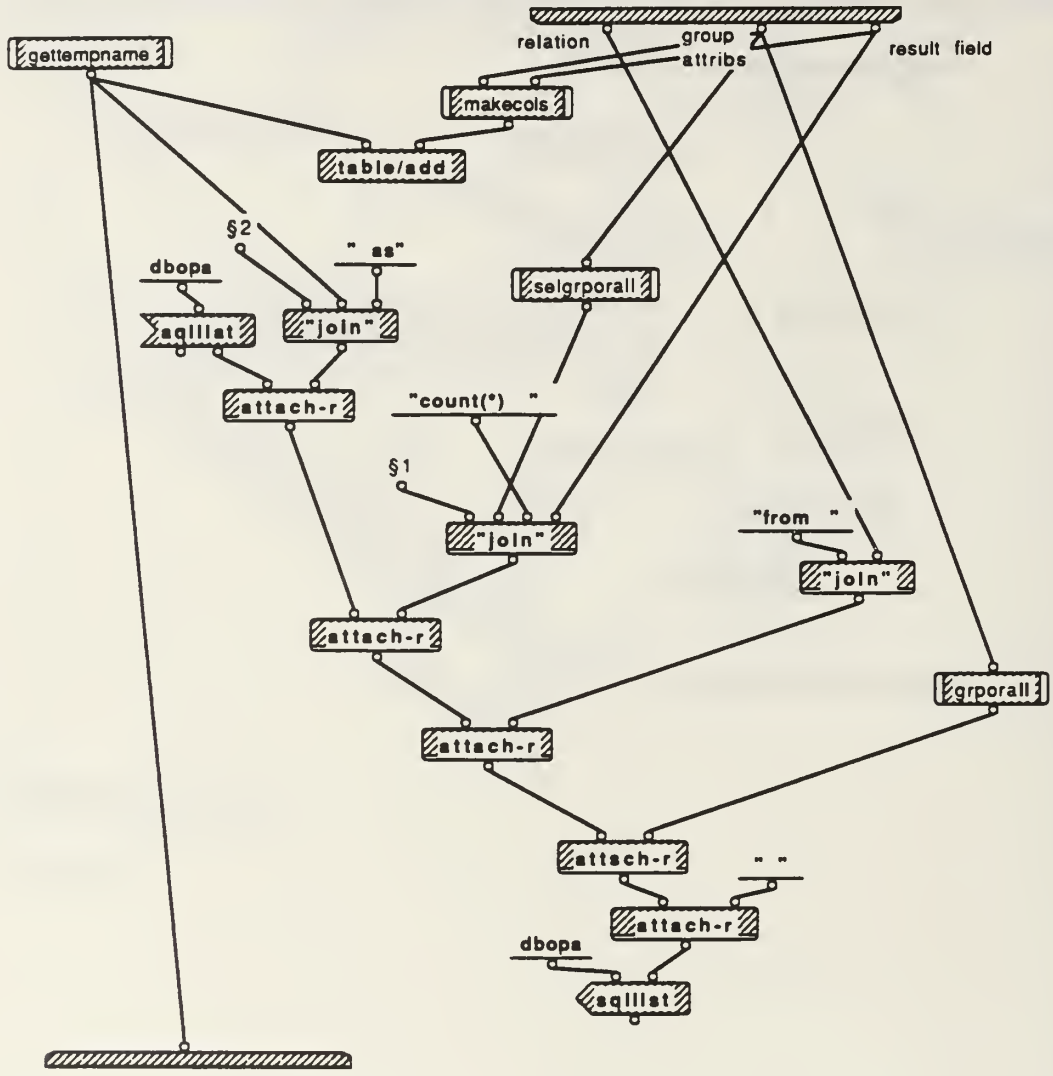


§1. "create view "

dbops/groupALLsatisfy 1:1

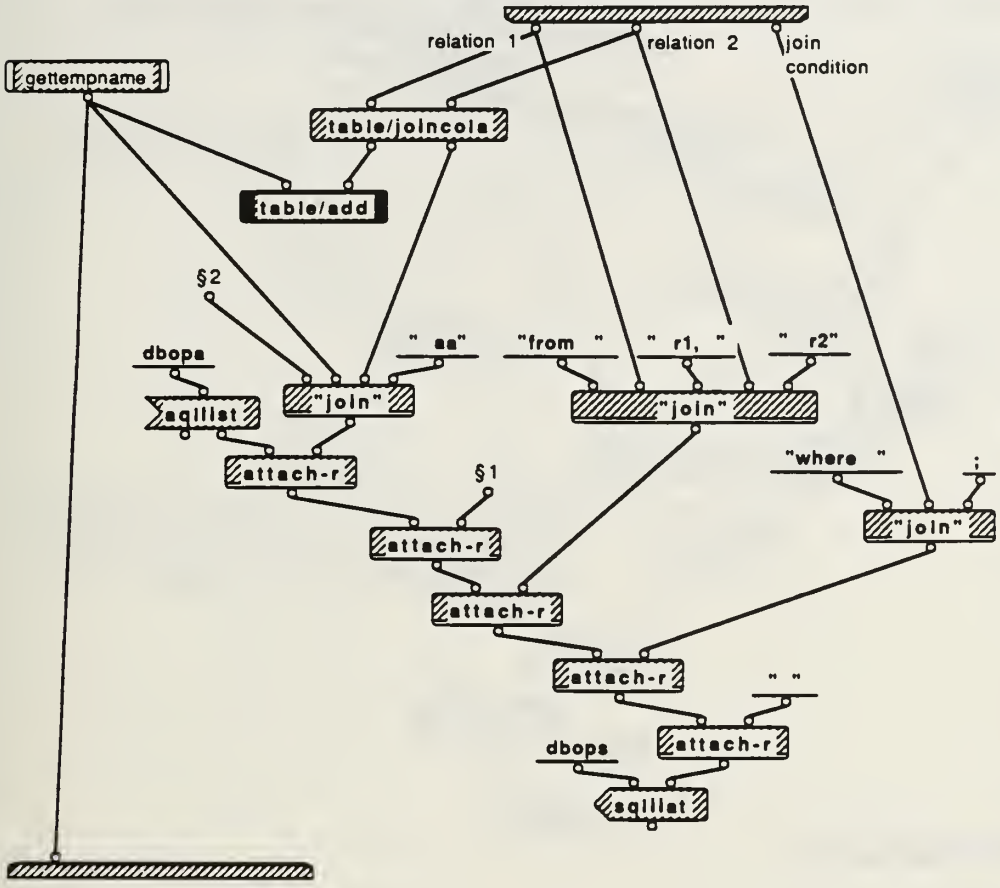


dbops/groupcnt 1:1



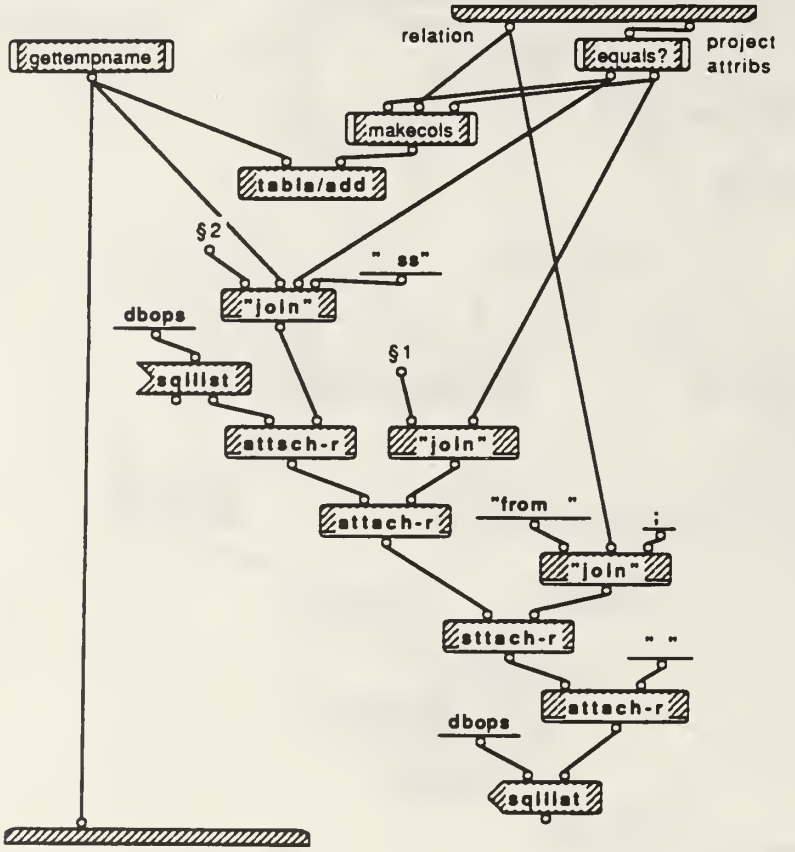
§1. "select distinct "
§2. "create view "

dbops/join 1:1



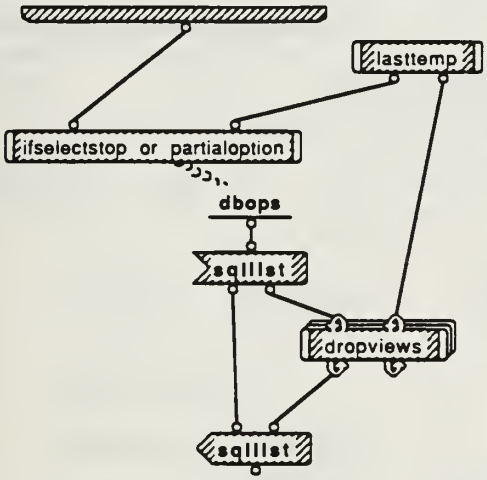
§1. select distinct *
§2. *create view *

dbops/project 1:1



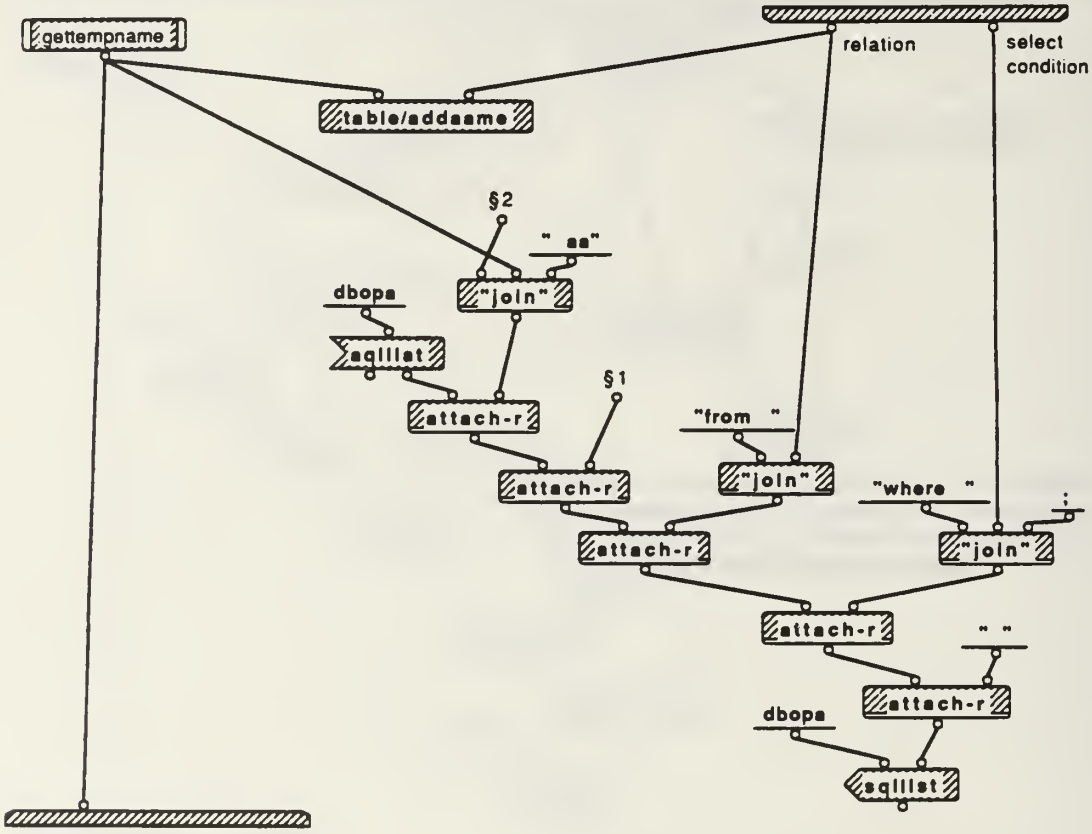
§1. "select distinct "
§2. "create view "

dbops/finalize 1:1



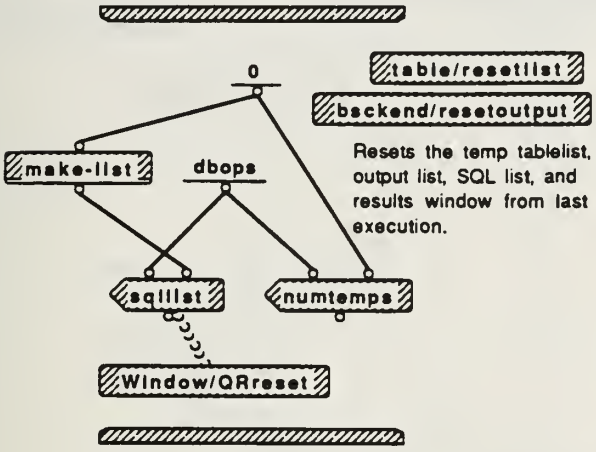
Adds SQL code to drop temporary views created by DFQL and display results if the Display Last option was selected OR if we stopped execution because of a selected operator.

dbops/select 1:1

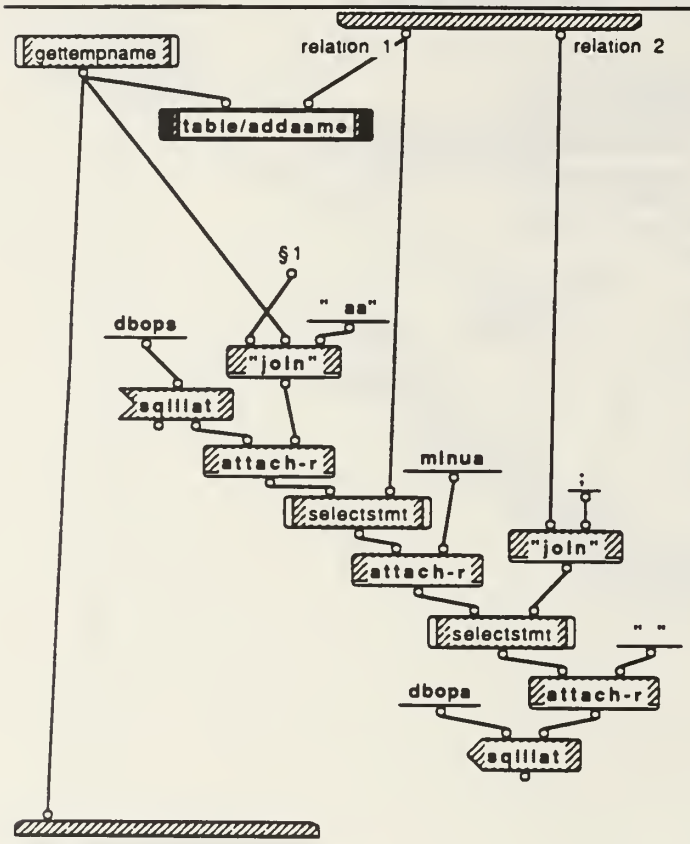


§1. select distinct *
§2. "create view "

dbops/reset 1:1

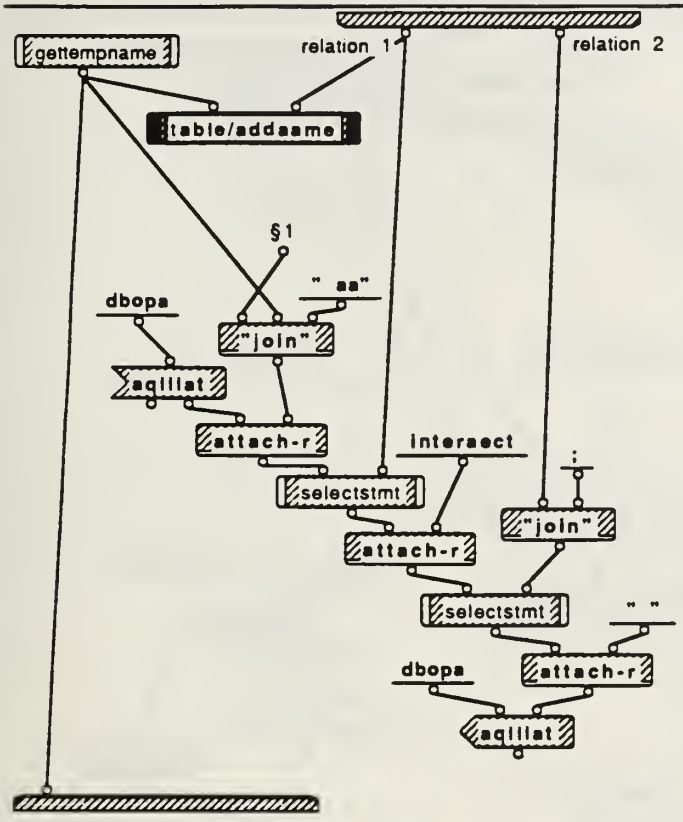


dbops/difference 1:1



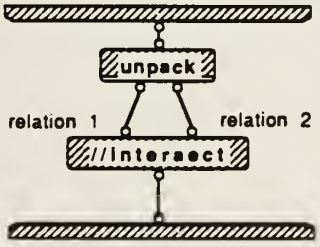
§1. "create view "

dbops/intersect 1:1

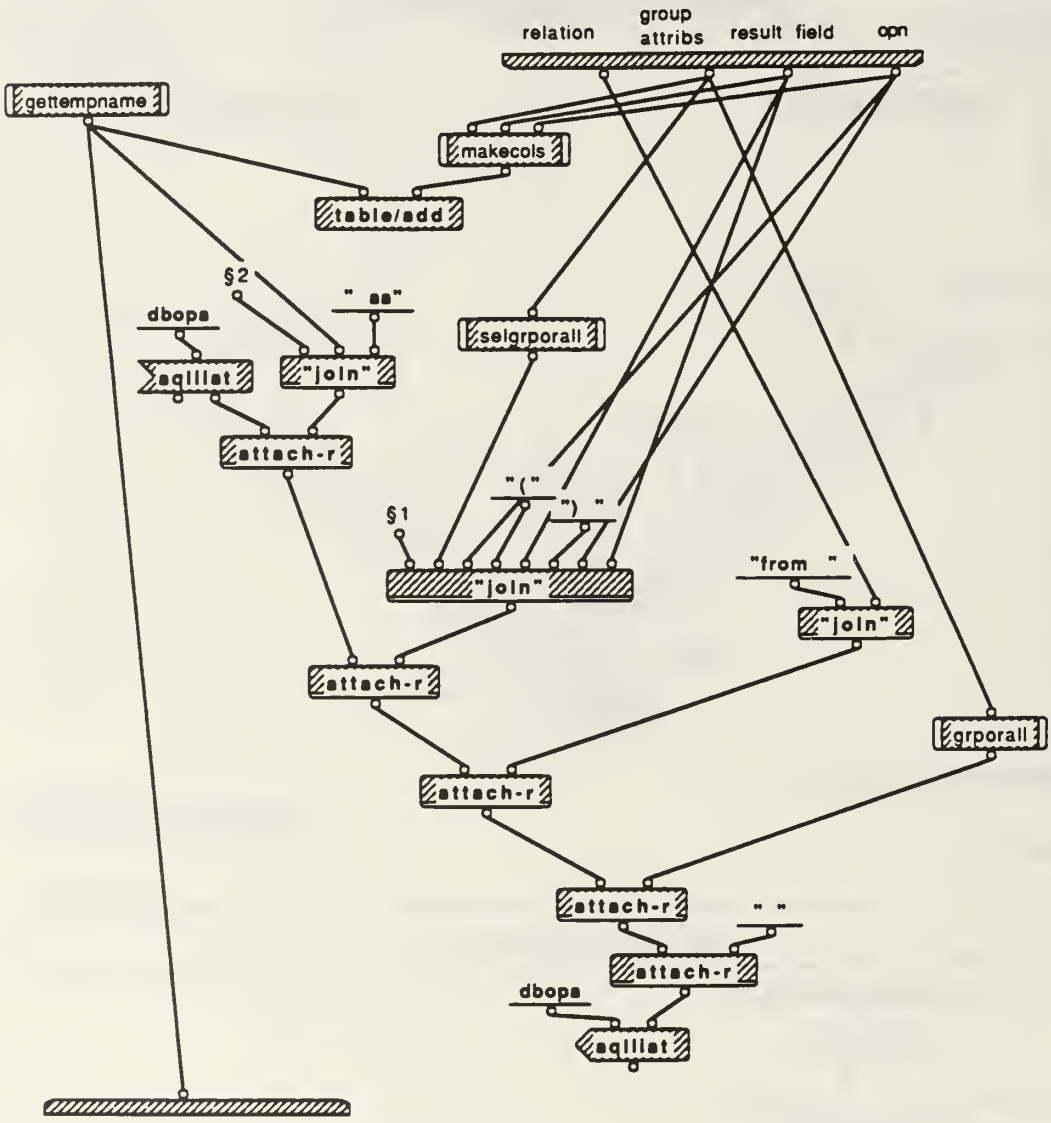


§1. "create view "

dbops/lintersect 1:1

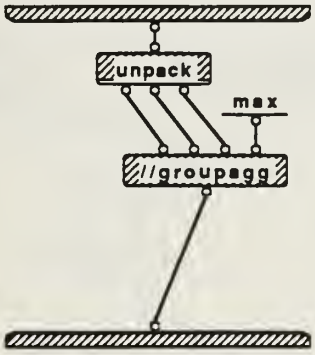


dbops/groupagg 1:1

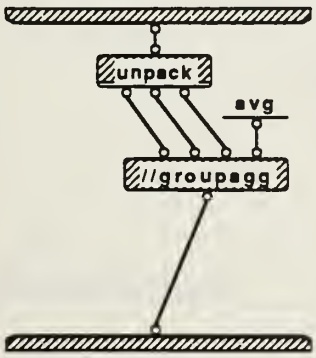


§1. "select distinct "
§2. "create view "

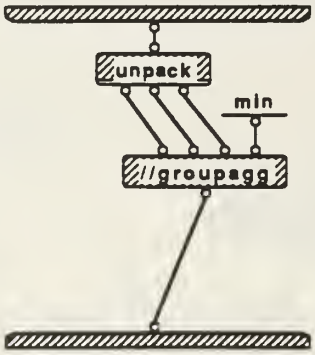
dbops/lgroupmax 1:1



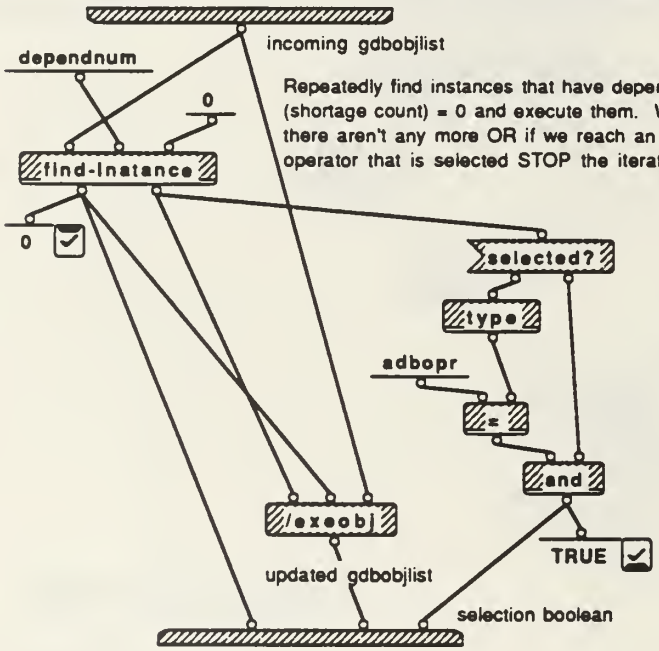
dbops/lgroupavg 1:1



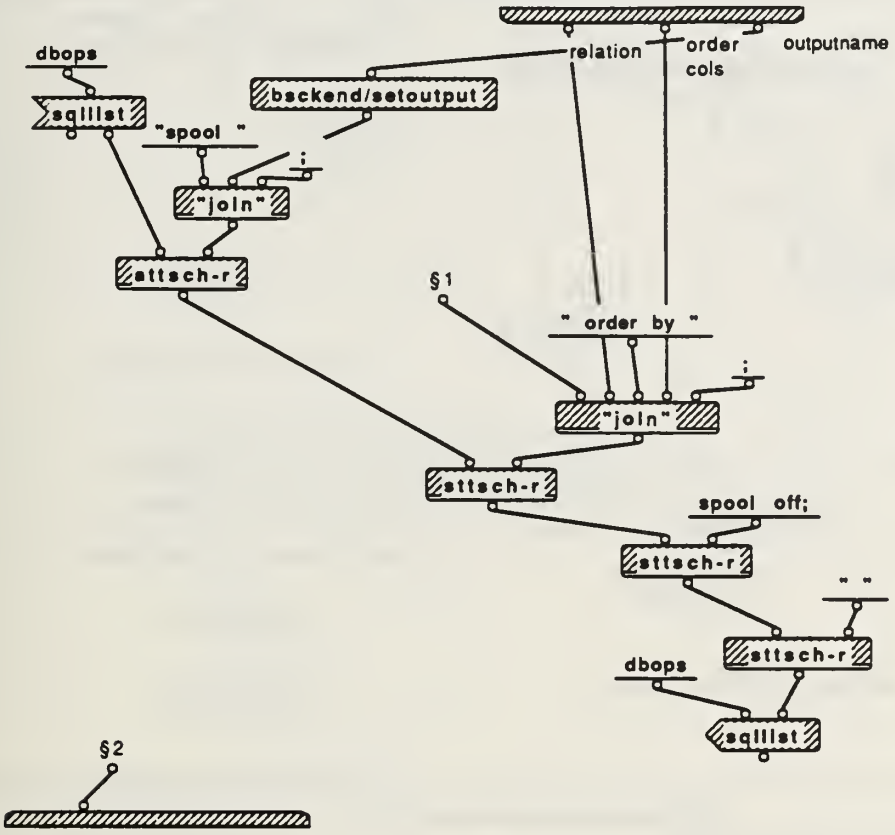
dbops/lgroupmin 1:1



dbops/doallops 1:1

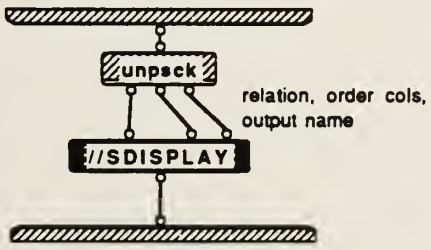


dbops/SDISPLAY 1:1

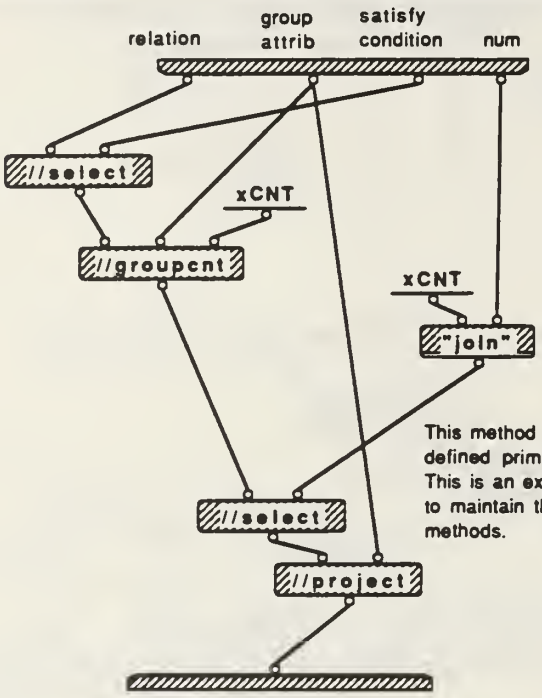


§1. "select distinct * from *
§2. Can't embed SORT in query!

dbops/ISDISPLAY 1:1

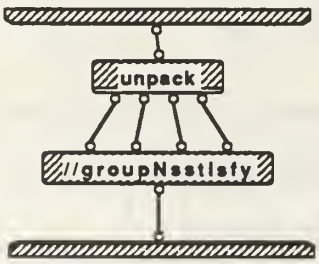


dbops/groupNsatisfy 1:1

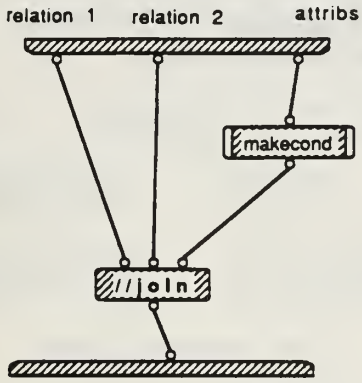


This method makes use of the other previously defined primitive methods to perform groupallsatisfy. This is an example of why we use the "l" methods to maintain the natural inputs to the DFQL primitive methods.

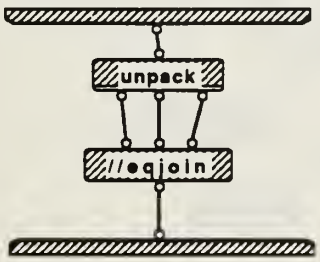
dbops/lgroupNsatisfy 1:1



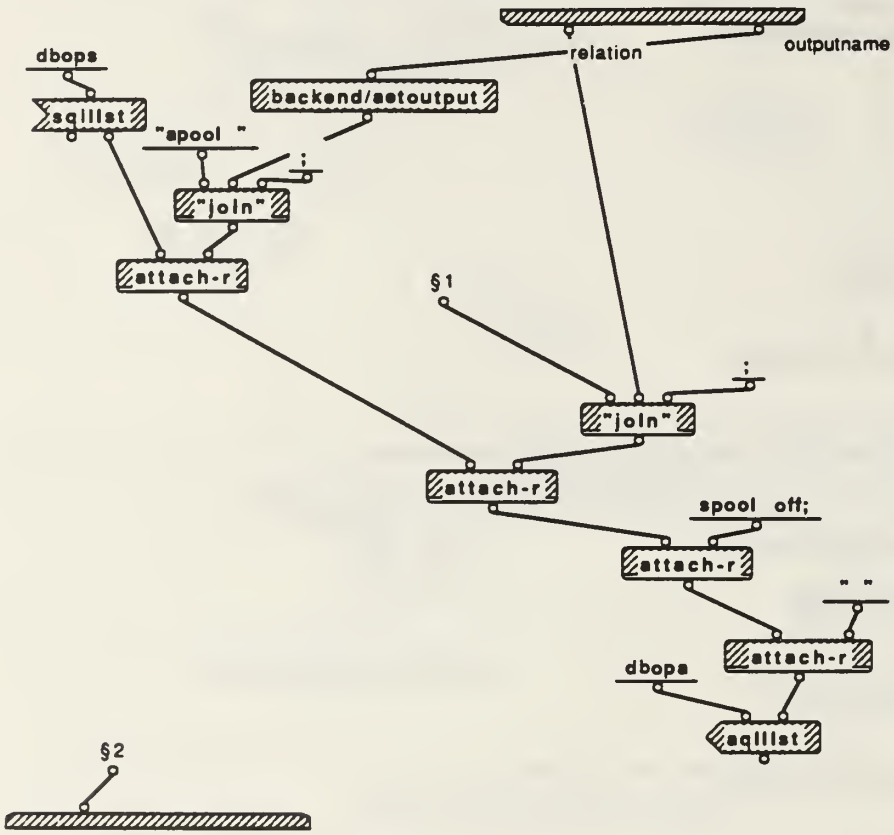
dbops/eqjoin 1:1



dbops/leqjoin 1:1

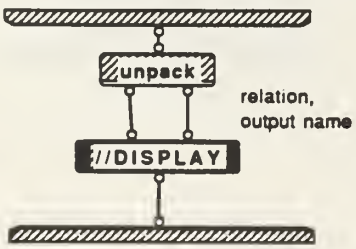


dbops/DISPLAY 1:1

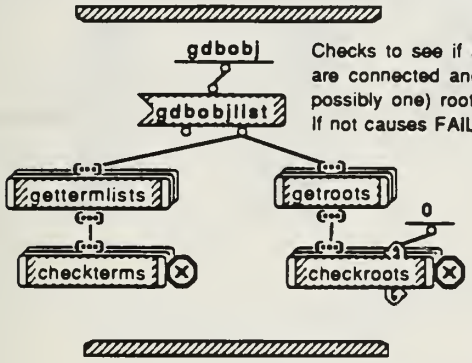


§1. "select distinct * from *
§2. Can't embed SORT in query!

dbops/IDISPLAY 1:1

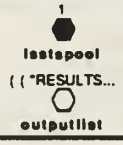


dbops/checkgraph 1:1



Checks to see if all terminals are connected and if all (except for possibly one) roots are connected. If not causes FAIL.

∇ backend



backend

ORACLE*Shell

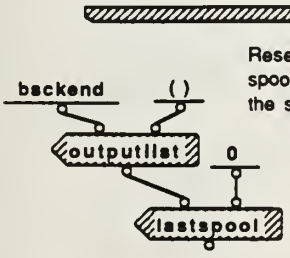
resetoutput

setoutput

runoracle

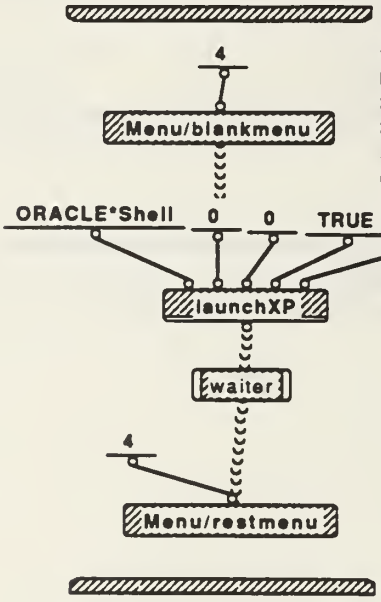
losdwinsql

backend/resetoutput 1:1



Resets the list of current spool files to empty and the spool file number to 0.

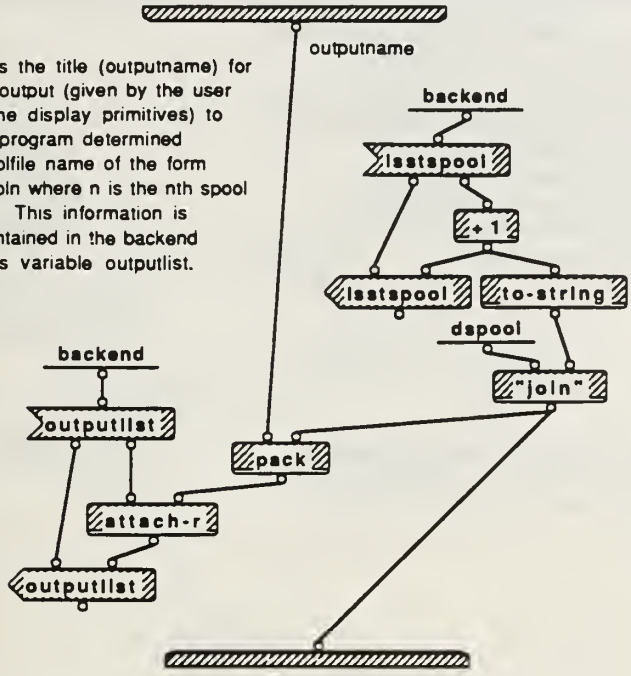
backend/ORACLE*Shell 1:1



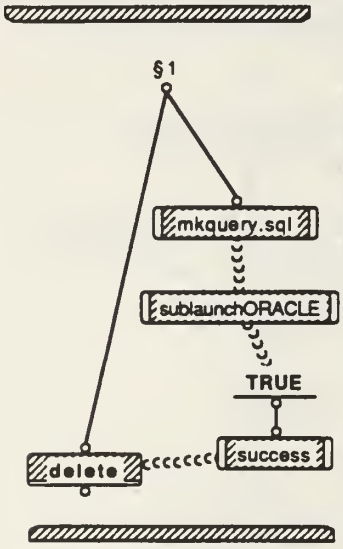
Sublaunches ORACLE*Shell to permit the user direct interaction with the backend database using ORACLES's SQL Plus Interpreter. This is a SUBLAUNCH -- the DFQL application is suspended during ORACLE*Shell operation.

backend/setoutput 1:1

Adds the title (outputname) for the output (given by the user in the display primitives) to the program determined spoolfile name of the form spoolN where n is the nth spool file. This information is maintained in the backend class variable outputlist.



backend/runoracle 1:1

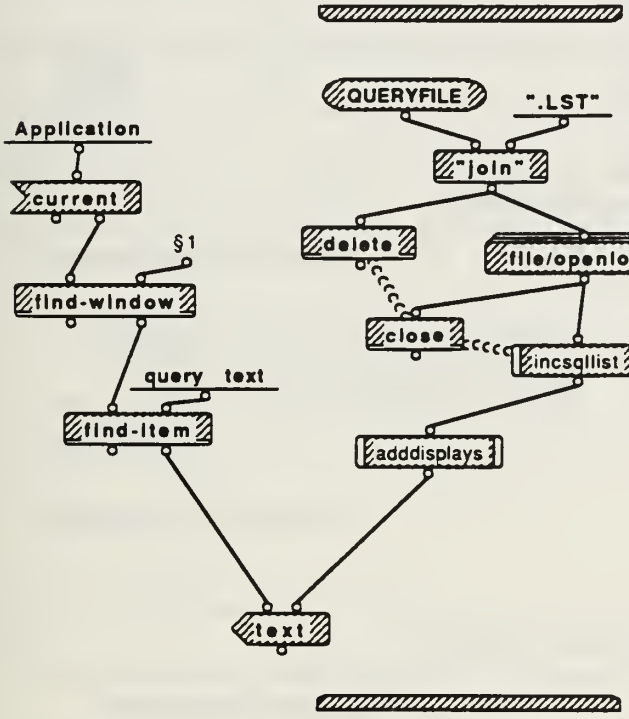


Writes the SQL query from sqllist into the file query.sql . ORACLE*Shell is then sublaunched in its batch mode to process query.sql . While ORACLE is running the DFQL program is suspended. ORACLE*Shell produces the query.sql.LST file and any spooled files that were requested by the users display statements.

Always set to true if control successfully returns. (Actually an artifact from an earlier implementation. Should be kept though.)

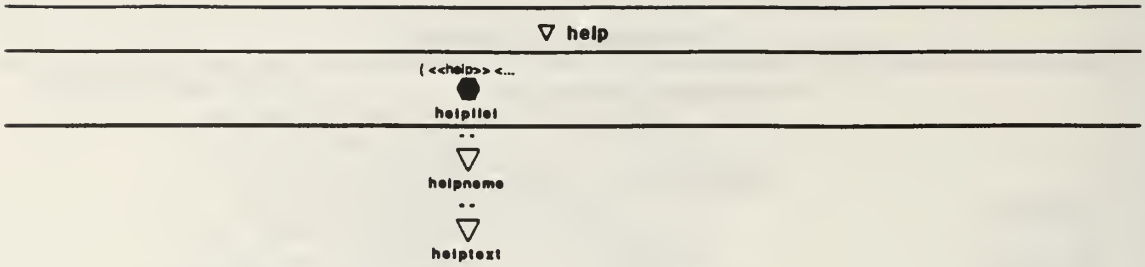
§1. N170i:PROGRAPH:query.sql

backend/loadwinsql 1:1



Loads results from all of the spool files requested by the user and also loads the entire contents of query.sql.LST (ORACLE*Shell's complete output) if the show SQL option was checked -- loads all of this into the Query Results window.

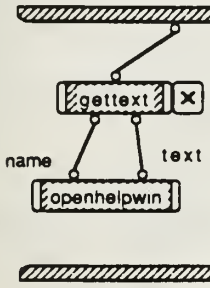
§1. Query Results


∇ help
(<<help>> <...
helpitel
helpneme
helptext

help



help/display 1:2

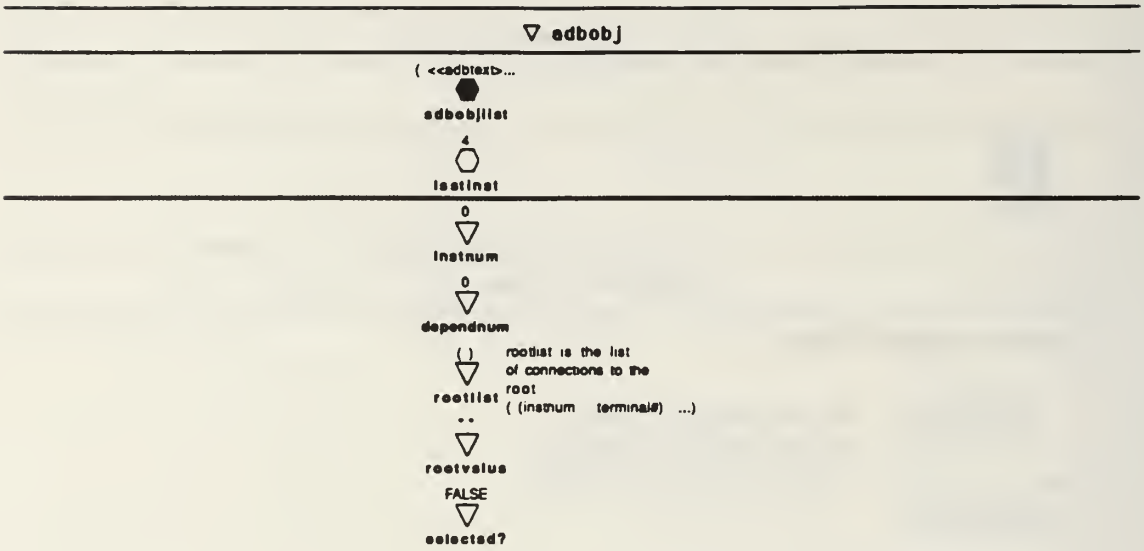


Gets help message from stored help list (primitives) or from the usropr (user-defined ops) and then opens the help window and displays it.

help/display 2:2



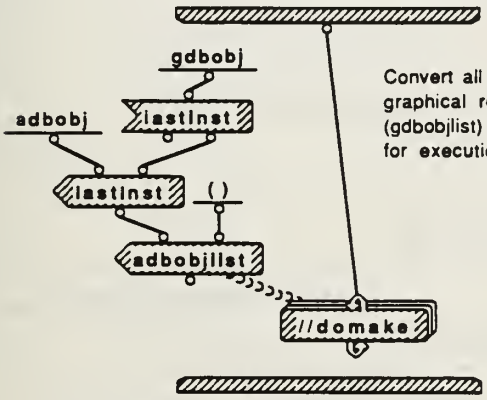
§1. Help will be available for this operator in the future!


▽ adbobj
(<<adbtext>...
adbobjlist
4
lastinst
0
instnum
0
dependnum
( )
rootlist
rootlist is the list of connections to the root
((instnum terminal#) ...)
• •
rootvalue
FALSE
selected?

adbobj

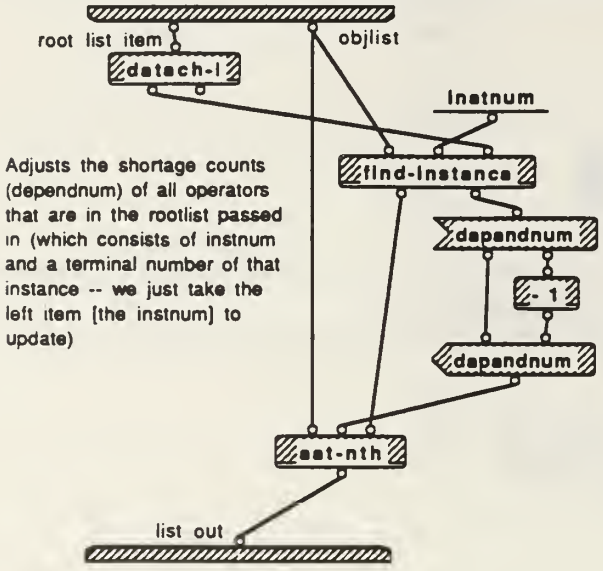


adbobj/makealist 1:1



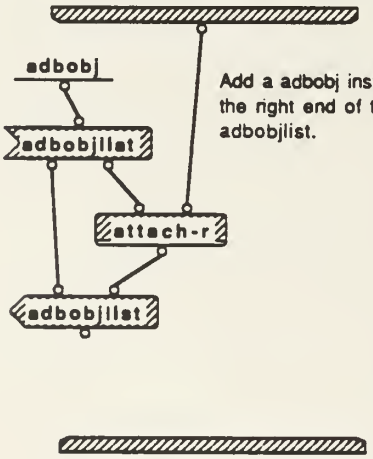
Convert all objects in the graphical representation (gdbobjlist) into adbobjs for execution.

adbobj/fixdependnums 1:1



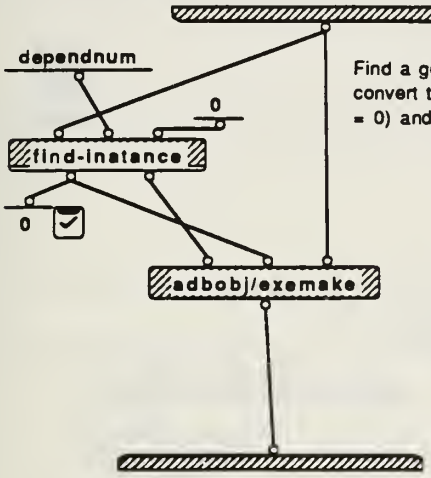
Adjusts the shortage counts (dependnum) of all operators that are in the rootlist passed in (which consists of instnum and a terminal number of that instance -- we just take the left item [the instnum] to update)

adbobj/addadbobj 1:1



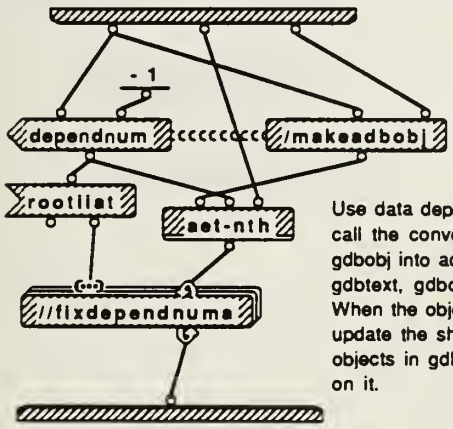
Add a adbobj instance to the right end of the adbobjlist.

adbobj/domake 1:1

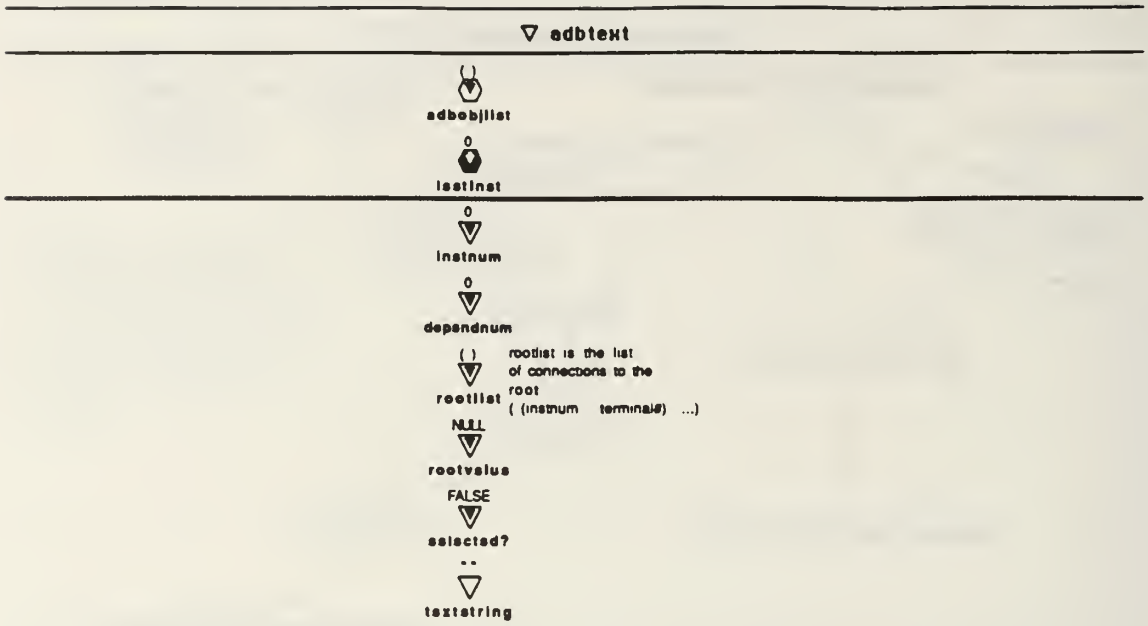


Find a gdbobj that is ready to convert to an adbobj (by dependnum = 0) and call exemake to convert it.

adbobj/exemake 1:1



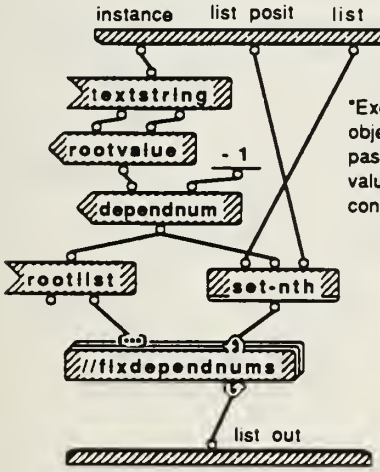
Use data dependent reference to call the conversion method from gdbobj into adbobj. This handles the gdbtext, gdbopr, and usropr objects. When the object has been converted update the shortage counts of the objects in gdbobjlist that depended on it.


∇ adbtext
adbobjlist
0
lastinst
0
instnum
0
dependnum
( )
rootlist
rootlist is the list of connections to the root
((instnum terminalß) ...)
NULL
rootvalue
FALSE
selected?
..
textstring

adbtext

exeobj

adbtext/exeobj 1:1



"Executes" the text objects by simply passing their text value on to the ops connected to it.

∇ adbopr

( )
adbobjlist

0
lastinst

0
instnum

NULL
dependnum

( )
rootlist
rootlist is the list of connections to the root
((instnum terminal#) ...)

NULL
rootvalue

FALSE
selected?

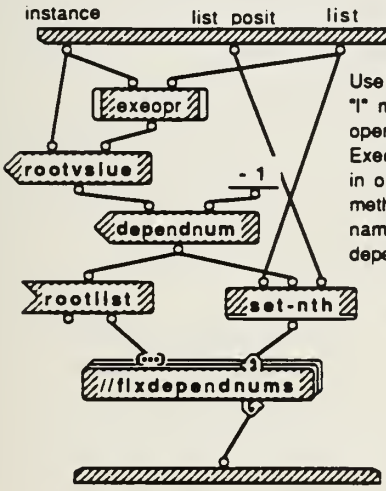
..
opnams

( )
terminallist
terminal connections to opr
((termrect from obj instnum) ...)

adbopr

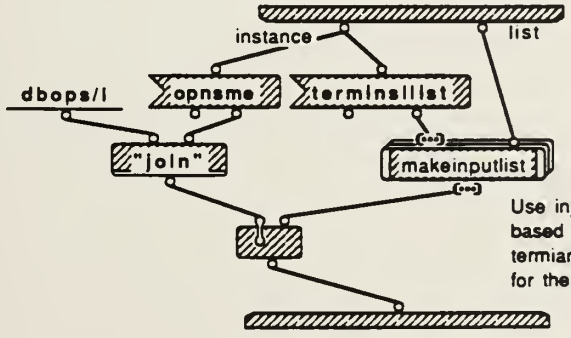
exeobj

adbopr/exeobj 1:1



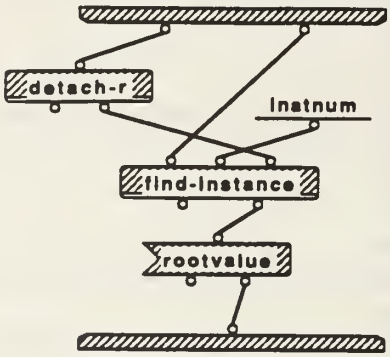
Use exeopr to call the dbops "I" method corresponding to the operator being executed. Exeopr uses the injection method in order to call the necessary method based on the DFQL oper name. After the op is executed dependnums are updated.

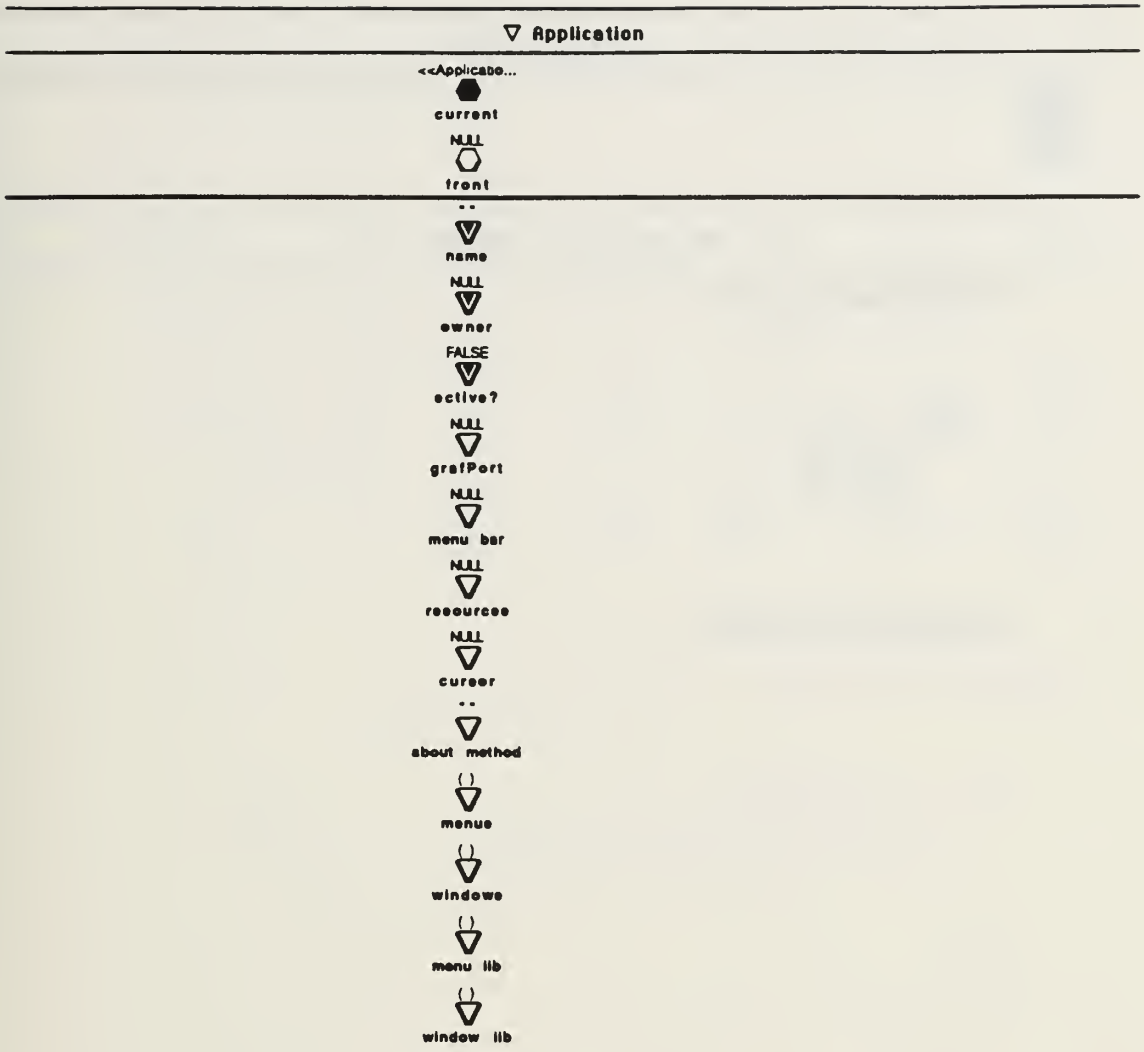
adbopr/exeobj 1:1 exeopr 1:1



Use injection to call the appropriate method based on the operator name. Take the ops termianal list and make it into a list of inputs for the actual dbops version of the DFQL operator.

adbopr/exeobj 1:1 exeopr 1:1 makeinputlist 1:1



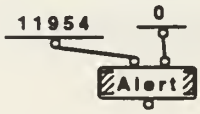

▽ Application
<<Applicabe...
current
NULL
front
..
name
NULL
owner
FALSE
active?
NULL
grafPort
NULL
menu bar
NULL
resources
NULL
cursor
..
about method
( )
menus
( )
windows
( )
menu lib
( )
window lib

Application



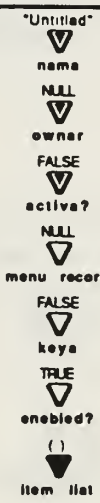
Application/about... 1:1







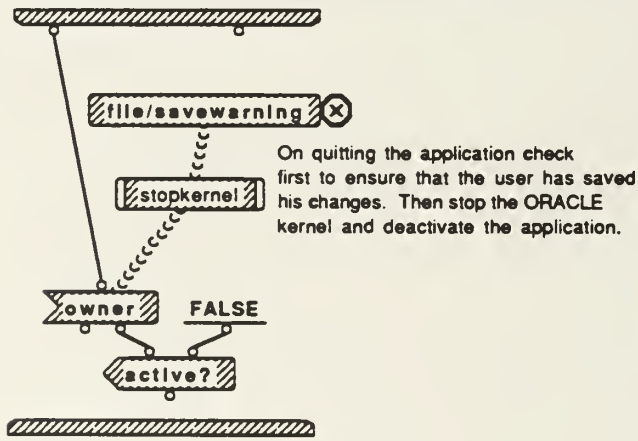
∇ Menu


"Untitled"
name
NULL
owner
FALSE
active?
NULL
menu_record
FALSE
keys
TRUE
enabled?
( )
item_list

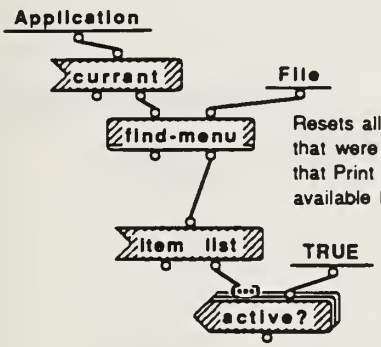
Menu



Menu/quit 1:1

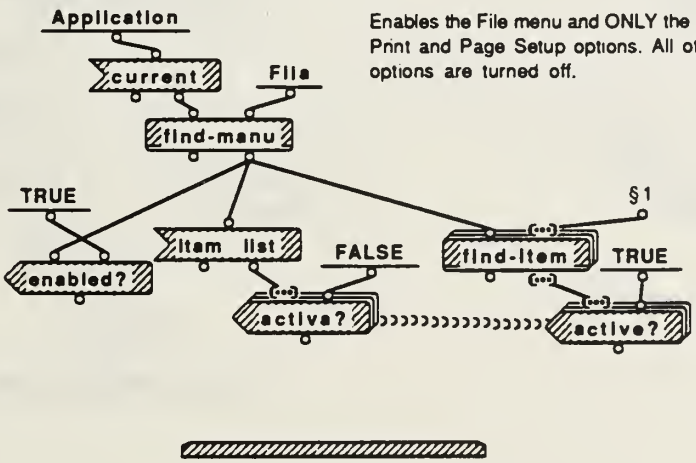


Menu/resetprt 1:1



Resets all of the items in the File menu that were turned off in certain modes so that Print and Page setup would be available but nothing else.

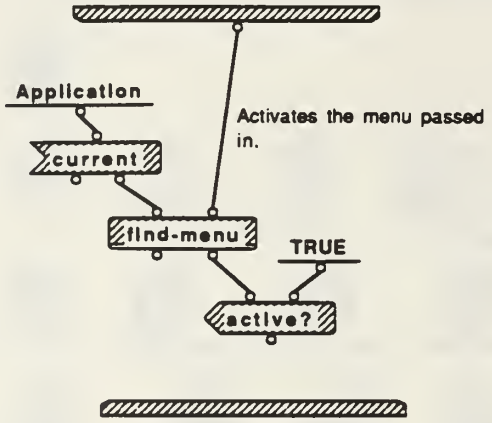
Menu/enableprt 1:1



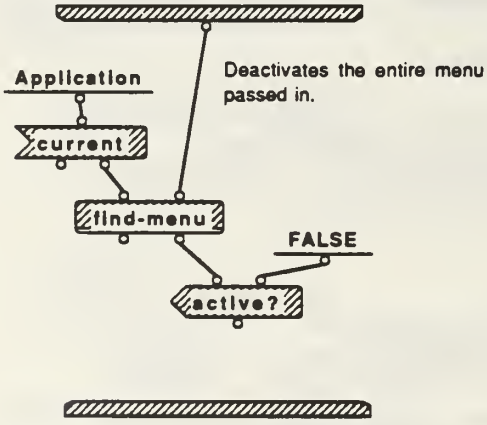
Enables the File menu and ONLY the Print and Page Setup options. All other options are turned off.

§1. ("Print..." "Page Setup...")

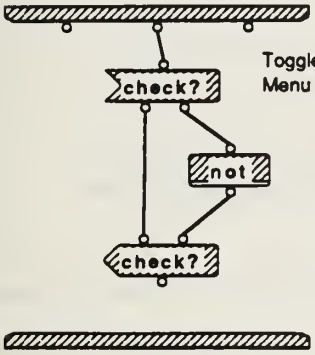
Menu/restmenu 1:1



Menu/blankmenu 1:1

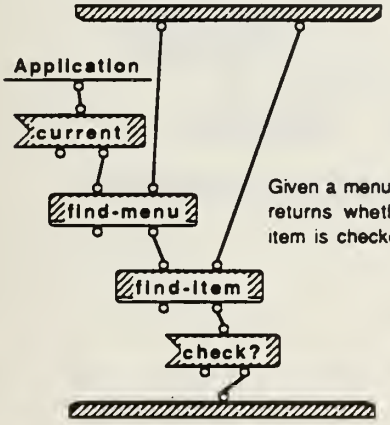


Menu/toggle 1:1



Toggles checkmark on Menu items.

Menu/itemcheck? 1:1



Given a menu and item returns whether the item is checked or not

Menu/doloadwinsql 1:1

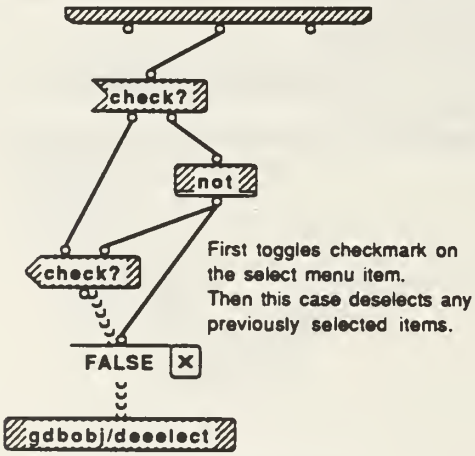




Called from the Edit menu
in response to the Undo item.
Located here in order to correctly
handle the inputs.



Menu/seltoggle 1:2



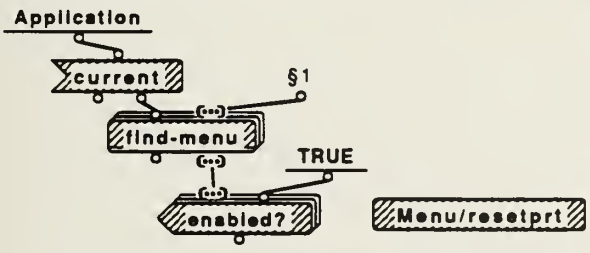
First toggles checkmark on
the select menu item.
Then this case deselects any
previously selected items.



Menu/seltoggle 2:2

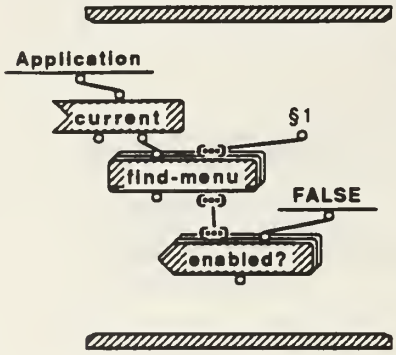
Menu/enable 1:1

Reenable all menus.



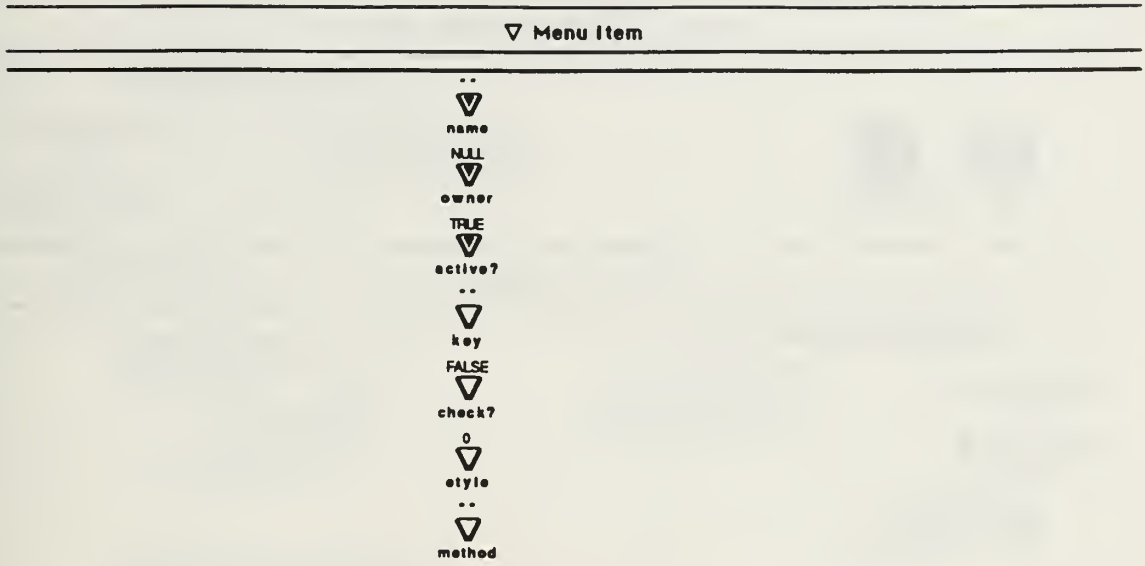
§1. (1 2 3 4 5 6 7)

Menu/disenable 1:1



Manually disenables all menus to simulate a true modal dialog for the help windows. Prograph's modal windows are not modal enough to turn off menu items.

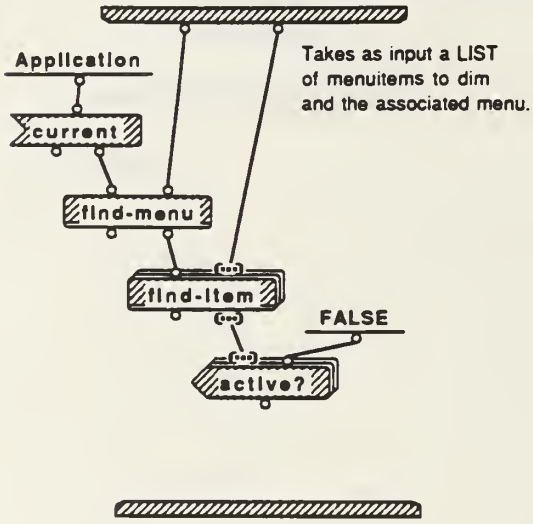
§1. (1 2 3 4 5 6 7)


▽ Menu Item
..
name
NULL
owner
TRUE
active?
..
key
FALSE
check?
0
style
..
method

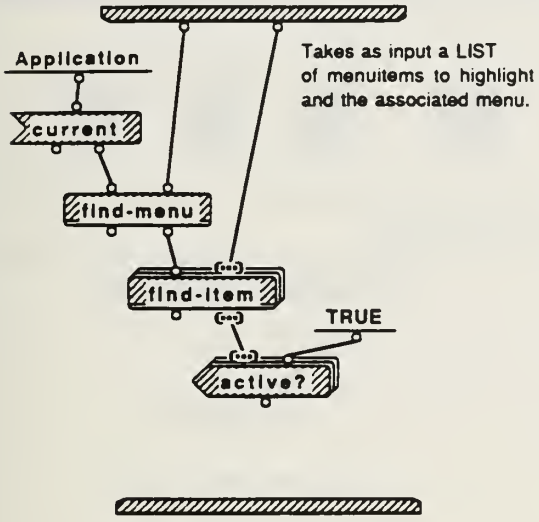
Menu Item

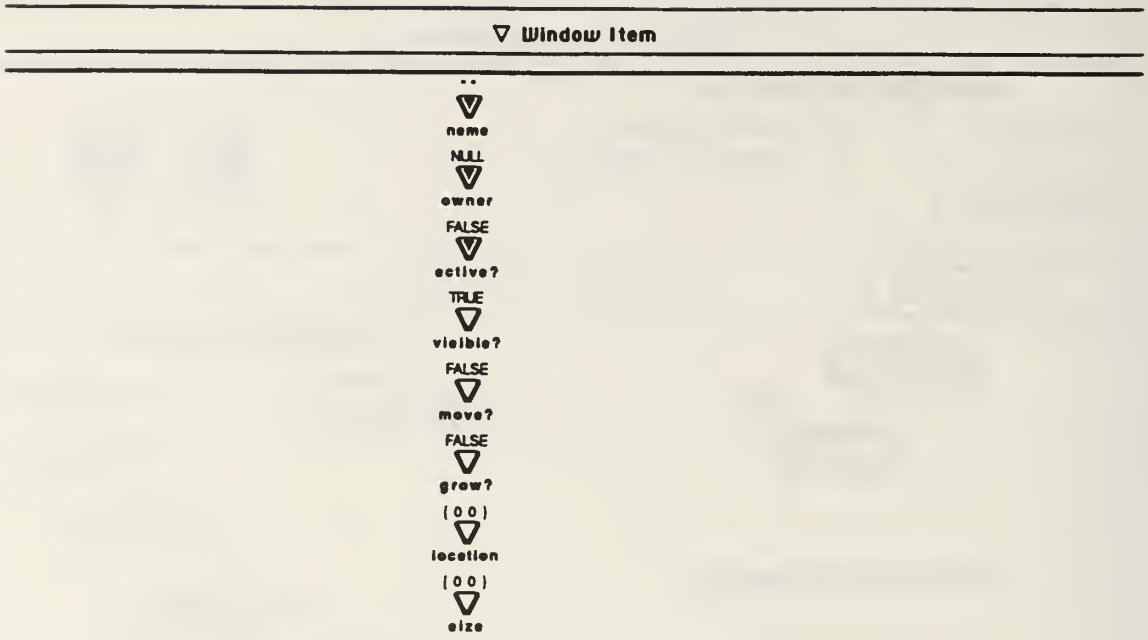


Menu Item/dim 1:1



Menu Item/highlight 1:1

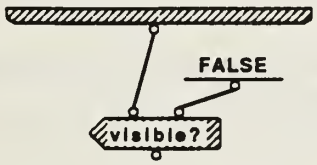



▽ Window Item
..
name
NULL
owner
FALSE
active?
TRUE
visible?
FALSE
move?
FALSE
grow?
{ 0 0 }
location
{ 0 0 }
size

Window Item

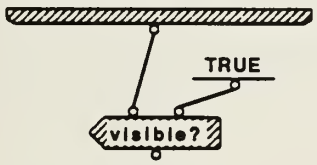


Window Item/visoff 1:1



Make a window item invisible.

Window Item/vison 1:1



Make a window item visible.

Window Item/activeoff 1:1

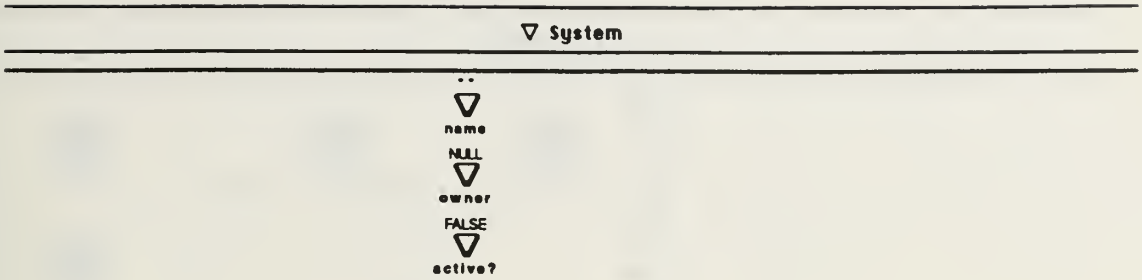


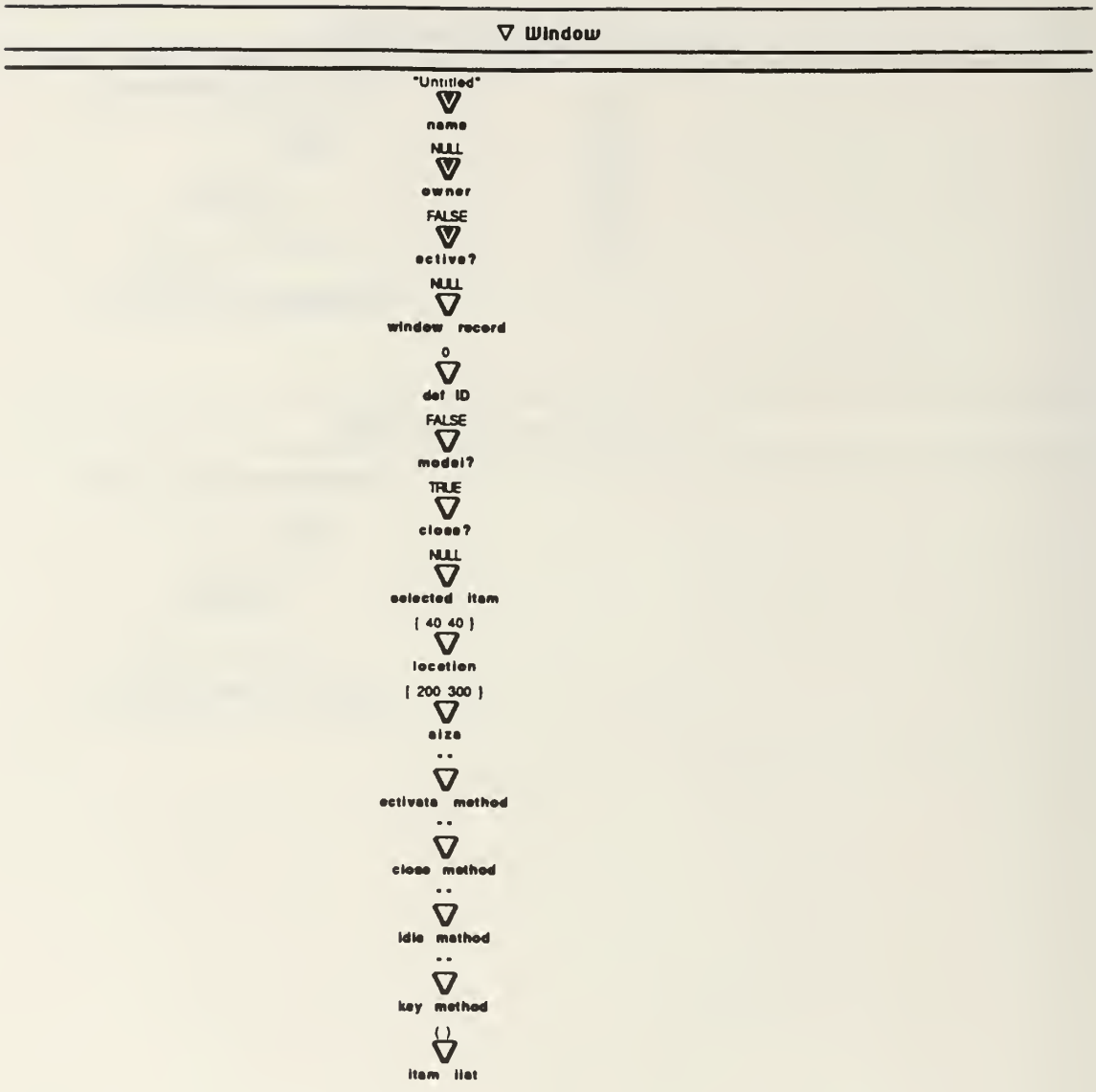
Dectivate a window item
(gray it out).

Window Item/activeon 1:1



Activate a window item.


∇ System
..
∇
name
NULL
∇
owner
FALSE
∇
active?


▽ Window
"Untitled"
name
NULL
owner
FALSE
active?
NULL
window record
0
def ID
FALSE
modal?
TRUE
close?
NULL
selected item
{ 40 40 }
location
{ 200 300 }
size
• •
activate method
• •
close method
• •
idle method
• •
key method
( )
item list

Universal

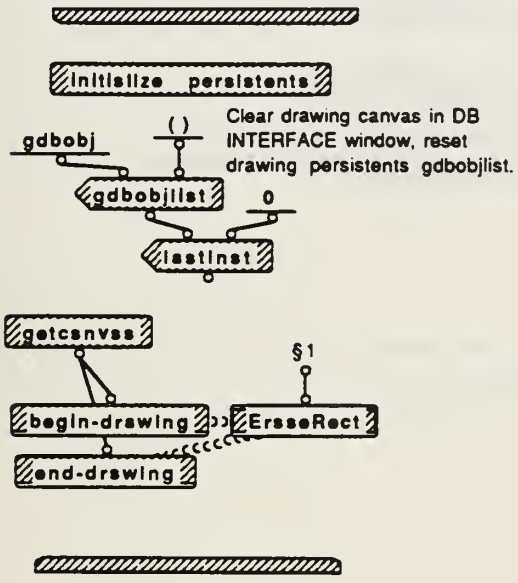
initial

initialize persistents

getcanvas

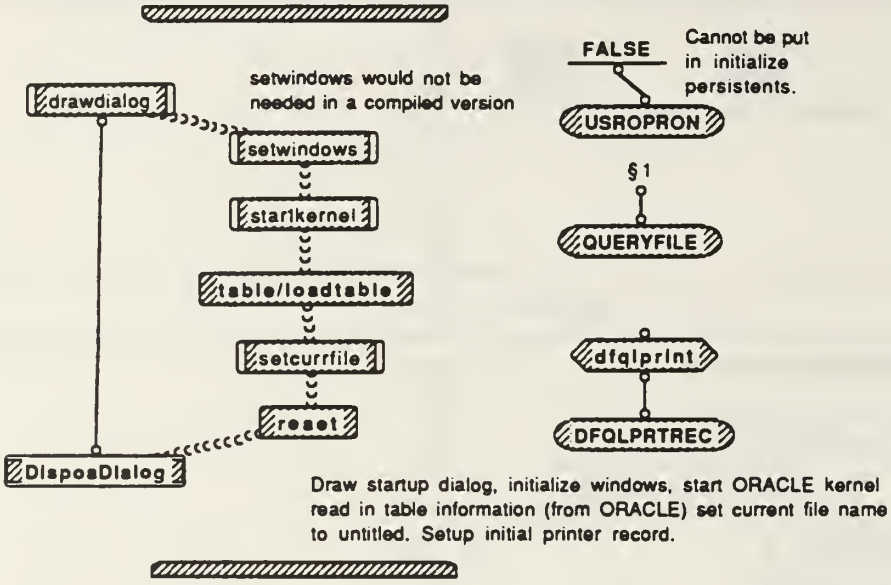
reset

reset 1:1



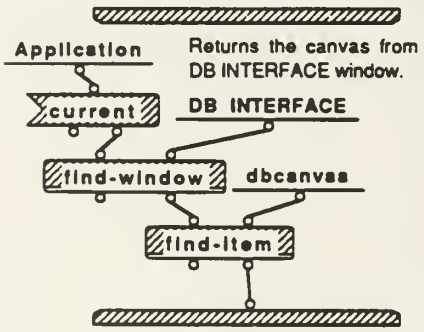
§1. {0 0 32000 32000}

initial 1:1

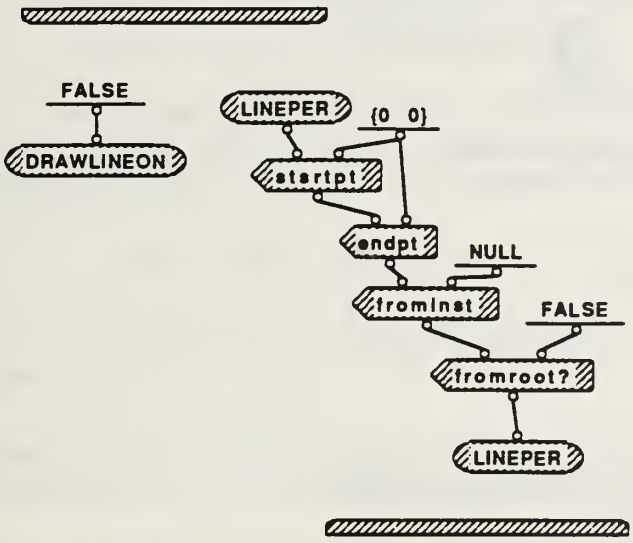


§1. "N170i:PROGRAPH:query.sql"

getcanvas 1:1



initialize persistents 1:1



Persistents



SQLSUCCESS

Success from last query.



QUERYFILE

Name of query file.



DRAWLINEON



LINEPER

Line drawing persistents to keep track of points for rubberband line.



DFQLPRTREC

Print record for printing DFQL canvases.



USROPRLIST



USROPRON

These six persistents are used to maintain user defined operator information, including the actual list of usroprs.



IBRECT



IBARLIST

List of INSTANCE nums of dummy text used for inbar roots.



DISPLAYOPR



TXTDISPLGTH

BIBLIOGRAPHY

Aho, A. V., and Ullman, J. D., "Universality of Data Retrieval Languages," *Proceedings of the Sixth ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pp. 110-120, 1979.

Beech, D., "The Future of SQL," *Datamation*, pp. 45-48, 15 February 1989.

Chandra, A. K., and Harel, D., "Computable Queries for Relational Databases," *Journal of Computer and System Sciences*, v. 21, pp. 156-178, 1980.

Dadashzadeh, M., "Improving Usability of The Relational Algebra Interface," *Journal of Systems Management*, pp. 9-12, September 1989.

Davis, J. S., "Usability of SQL and menus for database query," *International Journal of Man-Machine Studies*, v. 30, pp. 447-455, 1989.

Goodwin, N. C., "Functionality and Usability," *Communications of the ACM*, v. 30, pp. 229-233, March 1987.

INITIAL DISTRIBUTION LIST

1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Dudley Knox Library Code 052 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chairman, Computer Science Department Computer Science Department, Code CS Naval Postgraduate School Monterey, California 93943-5002	1
4.	Chief of Naval Research 800 North Quincy Street Arlington, Virginia 22217-5000	1
5.	Curriculum Officer Computer Technology Program, Code 37 Naval Postgraduate School Monterey, California 93943-5000	1
6.	Naval Ocean Systems Center 271 Catalina Boulevard San Diego, California 92152	1
7.	Division Head MDS Division Data Systems Department Naval Weapons Station Concord, California 94520-5000	1

8. Phillip B. Stiles — 1
Naval Sea Systems Command
Technical Data Division of the
Chief Engineer for Logistics Directorate
Washington, D. C. 20362-5101

9. James W. Hall — 1
Division Leader ADP Division
ADP-DO, MS-P222
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

10. C. Thomas Wu — 2
Computer Science Department, Code CSWq
Naval Postgraduate School
Monterey, California 93943-5000

11. LT Gard J. Clark — 2
484 Chestnut Road
Severna Park, Maryland 21146

Thesis
C48135 Clark
c.1 DFQL.

Thesis
C48135 Clark
c.1 DFQL.

DUDLEY KNOX LIBRARY
3 2768 00033070 8